

Fundamentos de
Informática

INFORMATICA PERSONAL-PROFESIONAL

RESPONSABLE EDITORIAL:
José Carlos Jiménez Pérez

AUTOEDICIÓN:
Sistemas de Imagen y Palabra

DISEÑO DE CUBIERTA:
Narcís Fernández

REALIZACIÓN DE CUBIERTA:
Gracia Fernández-Pacheco

Fundamentos de Informática

**Gregorio Fernández
Fernando Sáez Vacas**



Reservados todos los derechos. De conformidad con lo dispuesto en el artículo 534-bis del Código Penal vigente, podrán ser castigados con penas de multa y privación de libertad quienes reprodujeran o plagiaran, en todo o en parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte, sin la preceptiva autorización.

Edición española:

© EDICIONES ANAYA MULTIMEDIA, S.A., 1995

Juan Ignacio Luca de Tena, 15. 28027 Madrid

Depósito legal: S. 892-1995

ISBN: 84-7614-792-9

Printed in Spain

Imprime: Gráficas Varona

Polígono «El Montalvo», parcela 49

37008 Salamanca

Índice

Índice	5
Prólogo.....	19
Parte I	
Lógica	27
1. Ideas generales	29
1. Razonamientos y lógica formal	29
2. Lógica e informática.....	31
3. Lógica y circuitos de conmutación.....	32
4. Lógica, lenguajes y autómatas.....	32
5. Lenguajes formales: definiciones y conceptos básicos	33
5.1. Alfabeto, cadenas y lenguaje universal	33
5.2. Concatenación	34
5.3. Lenguaje y metalenguaje.....	34
5.4. Sintaxis, semántica y pragmática	35
6. Resumen	36

2. Lógica de proposiciones	39
1. Introducción.....	39
1.1. Variables proposicionales y sentencias	39
1.2. Conectivas	40
1.3. Interpretaciones y evaluaciones binarias de variables proposicionales y de sentencias	40
1.4. Ejemplos de formalización e interpretación	45
1.5. Cálculo, sistema axiomático y sistema inferencial	51
2. Sintaxis	53
2.1. Alfabeto	53
2.2. Expresiones y sentencias	53
2.3. Sentencias equivalentes	54
2.4. Conectivas primitivas y definidas	55
2.5. Axiomas, demostraciones y teoremas	56
3. Semántica.....	60
3.1. Interpretaciones y evaluaciones.....	60
3.2. Satisfacción.....	62
3.3. Tautologías y contradicciones	64
3.4. Completitud y consistencia de un sistema axiomático	64
3.5. Equivalencia entre sentencias.....	64
3.6. Las conectivas binarias.....	66
4. Modelo algebraico de la lógica de proposiciones.....	67
4.1. Algebra de Boole de las evaluaciones binarias	67
4.2. Algebra de Boole de las clases de equivalencia entre sentencias	69
4.3. Dos teoremas del álgebra de Boole	70
5. Sistemas inferenciales.....	71
5.1. Análisis y generación de razonamientos	71
5.2. Implicación lógica y razonamientos deductivos.....	72
5.3. Reglas de inferencia	73
5.4. Sistemas inferenciales	76
5.5. Forma clausulada de la lógica de proposiciones	78
5.6. Las cláusulas como sentencias condicionales	80
5.7. La regla de resolución	81
5.8. Refutación.....	85
6. Resumen	86
7. Notas histórica y bibliográfica.....	89
8. Ejercicios	90

3. Circuitos lógicos combinacionales.....	93
1. Introducción.....	93
2. Las puertas básicas	94
3. Circuitos	96
4. Modelos matemáticos de los circuitos.....	101
4.1. Utilidad de los modelos matemáticos.....	101
4.2. Funciones de conmutación	102
4.3. Formas booleanas	104
4.4. Relación entre los dos modelos matemáticos.....	108
5. Minimización.....	109
5.1. Principios.....	109
5.2. Método de Karnaugh.....	110
6. Ejemplos de aplicación.....	116
6.1. Máquina de escrutinio	116
6.2. Alarma para incendios.....	117
6.3. Etapa de sumador binario.....	119
6.4. Multiplicador binario de dos bits	121
6.5. Regulación de una piscina.....	123
7. La segunda forma canónica y la forma mínima en producto de sumas	125
7.1. La segunda forma canónica.....	125
7.2. La forma mínima en producto de sumas	127
8. Otras puertas.....	130
9. Circuitos con puertas "NAND" y "NOR".....	132
9.1. "NAND" y "NOR" como operaciones completas	132
9.2. La tercera forma canónica	133
9.3. La forma mínima sólo con NAND	133
9.4. La cuarta forma canónica	134
9.5. La forma mínima sólo con NOR	134
10. Resumen	134
11. Notas histórica y bibliográfica.....	136
12. Ejercicios	137
4. Lógica de predicados de primer orden.....	141
1. Introducción.....	141
1.1. Variables y constantes	141
1.2. Propiedades y relaciones	142
1.3. Predicados y fórmulas atómicas	142
1.4. Sentencias abiertas y cerradas	143

1.5. Cuantificadores.....	144
1.6. Evaluaciones binarias	144
1.7. Ejemplos de formalización e interpretación	146
1.8. Funciones.....	152
2. Sintaxis	152
2.1. Alfabeto	152
2.2. Términos y fórmulas atómicas	153
2.3. Sentencias	153
2.4. Sentencias abiertas y cerradas	154
2.5. Axiomas, demostraciones y teoremas	155
3. Semántica.....	158
3.1. Interpretaciones, asignaciones y evaluaciones	159
3.2. Satisfacción.....	161
3.3. Inconsistencia y validez.....	164
3.4. Modelos	164
4. Sistemas inferenciales.....	167
4.1. Razonamientos en lógica de predicados.....	167
4.2. Implicación lógica y razonamientos deductivos.....	167
4.3. Reglas de inferencia	169
4.4. Sistemas inferenciales	172
4.5. Forma clausulada de la lógica de predicados	172
4.6. Sustitución y unificación	176
4.7. La regla de resolución	178
4.8. Refutación.....	180
5. Resumen	182
6. Notas histórica y bibliográfica.....	182
7. Ejercicios	183
5. Otras lógicas.....	187
1. Introducción.....	187
2. Ampliaciones de la lógica de predicados	188
2.1. Lógica de predicados de orden superior.....	188
2.2. Lógica de clases y lógica de relaciones	189
2.3. Lógica de predicados con identidad	190
2.4. Lógica modal.....	191
2.5. Lógica temporal.....	193
3. Lógicas multivaloradas.....	193
4. Lógica borrosa	196
4.1. Justificación.....	196
4.2. Subconjuntos borrosos.....	198

4.3. Relaciones borrosas.....	201
4.4. Lógicas borrosas.....	206
5. Resumen	213
6. Notas histórica y bibliográfica.....	214
7. Ejercicios	215
6. Aplicaciones en ingeniería del conocimiento.....	219
1. Inteligencia artificial e ingeniería del conocimiento	219
1.1. ¿Qué es la inteligencia artificial?	219
1.2. La búsqueda heurística	221
1.3. Los sistemas expertos y los sistemas basados en conocimiento	221
1.4. La ingeniería del conocimiento	223
2. Sistemas basados en reglas.....	224
2.1. Sistemas de producción	224
2.2. Base de datos	225
2.3. Reglas de producción	225
2.4. Sistema de control (o motor de inferencias).....	227
2.5. Notas sobre el uso de la lógica de predicados. La "lógica 0+"	233
3. Inferencia plausible	235
3.1. Fuentes de imprecisión y de incertidumbre.....	235
3.2. Inferencia bayesiana	236
3.3. Inferencia mediante factores de certidumbre	242
4. Otros esquemas para la representación del conocimiento	245
5. Resumen	250
6. Notas histórica y bibliográfica.....	250
Parte II	
Autómatas.....	255
1. Ideas generales	257
1. Autómatas e información	255
2. Autómatas y máquinas secuenciales. Concepto de estado	257
3. Autómatas y lenguajes.....	257
4. Autómatas y álgebra	258
5. Resumen	258
2. Autómatas finitos.....	261
1. Definición y representación de los autómatas	261
1.1. Definición.....	261

1.2. Representación	262
1.3. Máquinas de Moore y de Mealy	264
2. Ejemplos de autómatas como modelos.....	267
2.1. Detector de paridad.....	267
2.2. Sumador binario serial.....	268
2.3. El castillo encantado.....	271
3. Comportamiento de un autómata.....	275
3.1. Otra definición de autómata	275
3.2. Ampliación del dominio de las funciones de un autómata.....	277
3.3. El comportamiento de entrada-salida, o las máquinas definidas por un circuito.....	277
3.4. Equivalencia y accesibilidad	278
3.5. Ejemplos	279
4. Capacidad de respuesta de un autómata finito.....	282
4.1. Introducción.....	282
4.2. Repaso de algunos conceptos de álgebra.....	283
4.3. Comportamiento de entrada-estados	288
4.4. Relación equirrespuesta y monoide de un autómata	290
4.5. Ejemplos	291
4.6. Relación entre el comportamiento de entrada-salida y el comportamiento de entrada-estados	295
5. Minimización de un autómata finito.....	297
5.1. Planteamiento del problema	297
5.2. Autómata en forma mínima de un autómata dado.....	297
5.3. Comprobación de la equivalencia entre estados de un autómata	298
5.4. Algoritmo para minimización de un autómata finito	300
5.5. Ejemplos	300
6. Resumen	303
7. Notas histórica y bibliográfica.....	303
8. Ejercicios	304
3. Circuitos secuenciales.....	309
1. La realización de autómatas finitos	309
2. Elementos de un circuito secuencial.....	310
2.1. Tipos de elementos	310
2.2. Elementos combinacionales	311
2.3. Elementos con memoria	311
3. Modelos básicos de circuitos secuenciales	316
4. Tipos de circuitos secuenciales.....	318
5. Análisis de circuitos secuenciales.....	319

6. Síntesis de circuitos secuenciales	320
6.1. Pasos de la síntesis	320
6.2. Ejemplos.....	322
7. Resumen	329
8. Notas histórica y bibliográfica.....	329
9. Ejercicios	329
4. Autómatas reconocedores y lenguajes regulares	333
1. Reconocedor finito	333
1.1. Definición.....	333
1.2. Ejemplos.....	334
2. Lenguajes aceptados por reconocedores finitos	336
2.1. Planteamiento del problema	336
2.2. Relación equirrespuesta de un reconocedor finito	337
2.3. Condición para que un lenguaje sea aceptado por un reconocedor finito	338
3. Conjuntos regulares y expresiones regulares	340
3.1. Los problemas de análisis y de síntesis.....	340
3.2. Conjuntos regulares.....	340
3.3. Expresiones regulares.....	341
4. Resolución de los problemas de análisis y de síntesis de un reconocedor finito.....	344
4.1. Análisis.....	344
4.2. Síntesis	346
5. Resumen	354
6. Notas histórica y bibliográfica.....	354
7. Ejercicios	355
5. Otros autómatas.....	359
1. Introducción.....	359
2. Redes de Petri	360
2.1. Estructura estática	360
2.2. Comportamiento dinámico	362
2.3. Autómata finito equivalente a una red de Petri con espacio de estados finito.....	365
2.4. Ejemplos de aplicación.....	366
3. Autómatas estocásticos.....	373
3.1. Definición.....	373
3.2. Ejemplo	374

3.3. Reconocedores estocásticos.....	376
4. Autómatas estocásticos de estructura variable	376
5. Autómatas con aprendizaje.....	377
5.1. Una primera definición de “aprendizaje”	377
5.2. Aprendizaje en un APEV	378
6. Autómatas borrosos	379
7. Autómatas celulares.....	382
8. Sistemas con aprendizaje.....	384
8.1. ¿Puede aprender una máquina?	384
8.2. Enfoques conductistas: redes neuronales	386
8.3. Enfoques cognoscitivos: adquisición de conceptos.....	388
8.4. Formación de conceptos y descubrimiento	391
9. Resumen	391
10. Notas histórica y bibliográfica.....	392

Parte III

Algoritmos	397
1. Ideas generales	399
1. Algoritmos y ordenadores	399
2. Algoritmos, lenguajes y programas	401
3. Algoritmos y máquinas de Turing	403
4. Computabilidad y complejidad.....	404
5. Resumen y conclusiones.....	406
2. Algoritmos	411
1. Introducción.....	411
2. Definiciones de algoritmo	411
3. Algoritmos y máquinas.....	413
3.1. El concepto de algoritmo, visto desde la teoría de conjuntos.....	413
3.2. Las máquinas, como estructuras capaces de ejecutar algoritmos	416
4. Propiedades de los algoritmos	417
4.1. Propiedad de finitud	417
4.2. Propiedad de definitud.....	417
4.3. Propiedad de generalidad	418
4.4. Propiedad de eficacia.....	418
5. Problemas sin algoritmo	418
6. Resumen	420

7. Notas histórica y bibliográfica.....	421
3. Programación estructurada.....	423
1. Introducción.....	423
2. Definición formal de un programa para ordenador	424
2.1. Definición algorítmica.....	424
2.2. Función, programa, función de programa	425
2.3. Normalización de organigramas.....	426
3. ¿Qué es un programa estructurado?	428
3.1. Estructuras básicas	428
3.2. Definición de programa estructurado	428
3.3. Teorema de estructura para un programa limpio.....	430
3.4. Ejemplo de aplicación del teorema de estructura	431
4. Método general de diseño de programas estructurados.....	432
4.1. Recursos abstractos	432
4.2. Razonamiento deductivo (diseño descendente)	433
4.3. Ejemplo de diseño de un programa estructurado	434
5. Programación estructurada en Pascal y C	437
5.1. Bloque múltiple	439
5.2. Bucle "for"	439
5.3. Instrucción CASE.....	439
6. Ventajas de la programación estructurada.....	440
7. Resumen	441
8. Notas histórica y bibliográfica.....	442
9. Ejercicios	442
4. Evolución de la programación.....	447
1. Introducción.....	447
2. Cómo se construye un programa	448
3. Técnicas de análisis y diseño.....	449
4. Programas correctos, robustos y convivenciales	451
5. Programación orientada a objetos	453
5.1. Propiedades de la programación orientada a objetos	456
5.2. Ventajas de la programación orientada a objetos.....	458
6. Análisis y diseño orientado a objetos	459
6.1. Metodologías de análisis y diseño.....	461
6.2. Ejemplo de análisis orientado a objetos	461
7. Resumen	464
8. Notas histórica y bibliográfica.....	465

5. Máquina de Turing: definición, esquema funcional y ejemplos	469
1. Introducción	469
2. Definición de máquina de Turing	470
3. Funcionamiento de la máquina de Turing a través de los ejemplos	472
3.1. Suma de dos números enteros no nulos escritos en el alfabeto $\{\}$..	472
3.2. Algoritmo de Euclides para el cálculo del m.c.d. de dos números enteros escritos en el alfabeto $\{\}$	476
3.3. Cálculo del m.c.d. de dos números enteros escritos en D ($D = \{0, 1, 2, 3, \dots, 9\}$) por el procedimiento general de construir un programa a base de subprogramas. Composición de máquinas de Turing	482
4. Diseño de una máquina de Turing	486
5. Simulación de máquinas de Turing, máquina de Turing universal y otras consideraciones	491
5.1. Simulación de máquina de Turing por ordenador	491
5.2. Máquina de Turing universal	496
5.3. Otras consideraciones	496
6. Sucédáneos de la máquina de Turing	498
7. Resumen	498
8. Notas histórica y bibliográfica	499
9. Ejercicios	500
6. Máquinas de Turing: algoritmos y computabilidad	507
1. Introducción	507
2. Función computable y función parcialmente computable	508
2.1. Hipótesis de Church o de Turing	508
2.2. Función computable	508
2.3. Ejemplos	509
3. Numerabilidad de la colección de todas las M.T.'s	510
3.1. Números de Gödel	510
3.2. Catálogo de las M.T.'s	512
4. De nuevo la máquina de Turing universal	512
4.1. Teorema	513
5. Conjuntos recursivos y recursivamente numerables	513
5.1. Conjunto recursivo	514
5.2. Conjunto recursivamente numerable	514
5.3. Dos teoremas más	515
6. Determinación de la finitud del proceso de cálculo. Problema de la aplicabilidad	515

7. Las máquinas de Turing y los lenguajes de tipo 0.....	516
8. Resumen	517
9. Notas histórica y bibliográfica.....	517
7. Complejidad	521
1. Introducción.....	521
2. Complejidad y máquinas de Turing.....	522
2.1. Dificultad de encontrar un criterio universal de complejidad	522
2.2. Complejidad espacial. Complejidad temporal.....	524
2.3. Máquinas deterministas, máquinas no deterministas y tipos de complejidad	526
2.4. Algunos resultados de la teoría de complejidad en máquinas de Turing	527
3. Medidas de la complejidad algorítmica.....	528
3.1. Complejidad polinómica y complejidad exponencial	529
3.2. Ejemplos de algoritmos con complejidad polinómica.....	530
3.3. Sinopsis.....	533
4. Problemas P, NP y NP-completos	536
4.1. El problema P-NP	536
4.2. Completitud y problemas NP-completos.....	537
4.3. Un tema no cerrado	538
5. Complejidad del software	538
5.1. Física del software	540
5.2. Número ciclomático	543
6. Resumen	544
7. Notas histórica y bibliográfica.....	545
8. Ejercicios	547
Parte IV	
Lenguajes.....	549
1. Ideas generales	551
1. Lenguajes e informática.....	551
2. Descripciones de los lenguajes	552
3. Sintaxis, semántica y pragmática.....	553
4. Dos ejemplos	553
5. Resumen	557

2. Gramáticas y lenguajes	559
1. Definición de gramática.....	559
2. Relaciones entre cadenas de E^*	560
2.1. Relación de derivación directa, \xrightarrow{G}	560
2.2. Relación de derivación, $\xrightarrow{*}_G$	560
3. Lenguaje generado por una gramática. Equivalencia de gramáticas	561
4. Ejemplos	561
5. Clasificación de las gramáticas y de los lenguajes	564
5.1. Gramáticas de tipo 0 ó no restringidas	564
5.2. Gramáticas de tipo 1 ó sensibles al contexto.....	564
5.3. Gramáticas de tipo 2 ó libres de contexto	565
5.4. Gramáticas de tipo 3 ó regulares	565
6. Jerarquía de lenguajes.....	565
7. Lenguajes con la cadena vacía.....	566
8. Resumen	568
9. Notas histórica y bibliográfica.....	568
10. Ejercicios	569
 3. De algunas propiedades de los lenguajes formales	 573
1. Introducción	573
2. No decrecimiento en las gramáticas sensibles al contexto	574
3. Recursividad de los lenguajes sensibles al contexto	576
4. Árboles de derivación para las gramáticas libres de contexto.....	576
5. Ambigüedad en las gramáticas libres de contexto.....	579
6. Resumen	584
7. Notas histórica y bibliográfica.....	584
8. Ejercicios	585
 4. Lenguajes y autómatas.....	 589
1. Introducción.....	589
2. Lenguajes de tipo 0 y máquinas de Turing.....	590
2.1. Reconocedor de Turing	590
2.2. Lenguaje aceptado por un reconocedor de Turing	591
2.3. Teorema MT1	591
2.4. Teorema MT2.....	591
2.5. Conclusión.....	591
3. Lenguajes sensibles al contexto y autómatas limitados linealmente	592
3.1. Autómata limitado linealmente	592
3.2. Lenguaje aceptado por un reconocedor limitado linealmente	592

3.3. Teorema ALL1	592
3.4. Teorema ALL2	593
3.5. Conclusión	593
4. Lenguajes libres de contexto y autómatas de pila	593
4.1. Autómata de pila.....	593
4.2. Lenguaje aceptado por un reconocedor de pila	595
4.3. Teorema AP1.....	595
4.4. Teorema AP2.....	595
4.5. Conclusión.....	595
5. Lenguajes regulares y autómatas finitos.....	595
5.1. Autómata finito no determinista	595
5.2. Lenguaje aceptado por un reconocedor finito no determinista, y equivalencia con algún reconocedor finito determinista.....	598
5.3. Teorema AF1.....	601
5.4. Teorema AF2.....	604
5.5. Conclusión.....	605
5.6. Observaciones sobre la cadena vacía	605
6. Jerarquía de autómatas.....	606
7. Resumen	607
8. Notas histórica y bibliográfica.....	607
9. Ejercicios	608
5. Aplicaciones a los lenguajes de programación.....	611
1. Lenguajes, traductores e intérpretes	611
2. Lenguajes de programación.....	614
2.1. Niveles de lenguajes.....	614
2.2. Lenguajes de bajo nivel	615
2.3. Lenguajes de alto nivel.....	621
2.4. Lenguajes declarativos	625
2.5. Entornos de desarrollo.....	628
3. Definiciones sintácticas	629
3.1. Notación de Backus.....	629
3.2. Gramáticas para lenguajes de programación.....	631
3.3. Gramática para un lenguaje ensamblador.....	633
3.4. Gramática para un lenguaje de alto nivel	636
3.5. Diagramas sintácticos.....	640
4. Definiciones semánticas	643
4.1. Objetivos.....	643
4.2. Semántica operacional.....	645
4.3. Semántica denotacional	647

4.4. Semántica axiomática	650
5. Procesadores de lenguajes	652
5.1. Ensambladores.....	652
5.2. Compiladores.....	661
5.3. Intérpretes	681
5.4. Procesadores de lenguajes declarativos.....	683
5.5. Herramientas para la generación de procesadores de lenguaje	685
6. Resumen	686
7. Notas histórica y bibliográfica.....	688
Referencias bibliográficas	691
Índice alfabético	713

Prólogo

"Nada permanece, todo cambia", decía Heráclito. Incluso este libro, que, por su carácter básico, tiene vocación de permanencia, ha sufrido dos importantes tipos de cambio.

El primero de ellos es que, por razones empresariales, ha cambiado de sello editorial. Antes, desde 1987, era conocido en el mundo universitario por el título de "Fundamentos de Informática" y publicado en la colección Alianza Informática, de Alianza Editorial. Agotada dicha edición, publicamos ahora un nuevo libro, bajo el sello de Anaya Multimedia, una división del Grupo Anaya, como también lo es actualmente Alianza Editorial.

El segundo, y más importante para sus lectores precedentes y para los nuevos, lo constituye la renovación de sus contenidos, con la inclusión de algunos capítulos y secciones, la remodelación profunda de otros, la actualización de las referencias bibliográficas y diversos retoques por aquí y por allá, cambios orquestados para mantener por un lado las esencias que lo han afianzado como texto de referencia en numerosos programas de la universidad y, por otro, para modernizarlo señalando a sus lectores las vías de las aplicaciones más avanzadas. Estimamos que estos cambios representan en volumen, no en extensión, aproximadamente un 20% del total del libro ya desaparecido. Todo ello se explica con mayor detalle en la segunda parte de ese prólogo. Su primera parte, que podrán leer a renglón seguido, es una reproducción del prólogo del libro anterior, que en buena parte sigue siendo éste.

Este libro trata exactamente de lo que dice su título.

Como primera impresión, a algunas personas puede parecerles innecesario publicar un texto sobre fundamentos en una época en que la informática ha llegado

en su difusión casi a formar parte material del mobiliario hogareño. Además de este efecto distributivo sobre la sociedad, los espectaculares progresos tecnológicos han producido un crecimiento desbordante de la especialización. Los sistemas operativos, las bases de datos, los lenguajes concurrentes, la programación lógica, la inteligencia artificial, la arquitectura de ordenadores, las redes, las herramientas de ayuda para ingeniería de software y tantas otras más específicas e instrumentales son áreas de trabajo o técnicas que por sí solas requieren esfuerzos considerables por parte de quienes pretenden estudiar y seguir su evolución.

Nadie discute que la especialización es inevitable. También es verdad que los inconvenientes que genera su abuso dentro de muchos de los dominios de aplicación de la informática son importantes y cada día somos más los que pensamos que aquéllos tienden a crecer en la misma proporción en que el especialista ignora la perspectiva global donde se integra su particular disciplina. Con mayor razón, si sucede que además desconoce las raíces de sus técnicas profesionales. Es decir, la proliferación de especialistas demasiado pragmáticos contribuye a crear una situación que comienza a ser ya preocupante y contra la que se han alzado voces diversas. Estas voces se muestran unánimes en apelar a la necesidad de construir o reconstruir una formación más sólida y básica en informática, dirigida como mínimo a aquéllos que se sienten deseosos de comprender profundamente su profesión o de implicarse en un papel técnico significativo con independencia de la especialidad escogida (por no hablar de la lucha contra la obsolescencia técnica que acecha a todos cuantos trabajamos en este campo).

El libro que presentamos responde a esta llamada, puesto que se ocupa de algunas de las raíces o fundamentos de la informática. Lo hemos escrito pensando en los estudiantes universitarios de las ramas de informática, así como en los profesionales antes mencionados. Estos últimos encontrarán un texto autocontenido, desprovisto en lo posible del aparato teórico habitual y preocupado permanentemente en la tarea de desarrollar aperturas a cuestiones de la más viva actualidad, como los sistemas borrosos o la complejidad del software, y a cuestiones en las que parece vislumbrarse un futuro. En cuanto a los estudiantes, nuestra experiencia nos dice que, por un cúmulo de circunstancias que no hacen al caso, se ven obligados con frecuencia a estudiar las materias objeto de nuestro libro, tal vez, sí, con mayor extensión y formalismo matemático, pero no siempre bajo condiciones óptimas: apuntes improvisados, textos en lenguas extranjeras, dispersión de estas mismas materias en distintas asignaturas y por tanto fragmentación de su sentido radical (raíces), o desapego del sentido de su aplicación. Sin poner en tela de juicio la necesidad científica del mejor formalismo posible, está constatado que dosis excesivas y exclusivas de esa medicina conducen en el plano educativo a un estéril desánimo de los estudiantes.

Consta el libro de cuatro partes, centradas en cuatro de los pilares básicos sobre los que se sustenta el edificio teórico y práctico de la informática. Estos son la lógica, la teoría de autómatas, la teoría de la computación y la teoría de los lenguajes formales. Por razón del enfoque adoptado, mixto entre teoría y práctica,

y por razones de concisión, aquí los hemos enmarcado bajo los escuetos rótulos de "Lógica", "Autómatas", "Algoritmos" y "Lenguajes". Para las referencias cruzadas entre partes del texto se decidió utilizar el vocablo *tema* en lugar de *parte*. Así, mencionaremos el apartado tal del capítulo cual del tema de "Lógica", y no de la *Primera Parte* (por ejemplo).

A muy grandes rasgos, podemos esbozar el origen o motivación inicial de cada área: filosófico (naturaleza del razonamiento humano) para la lógica, tecnológico (formalización del diseño de circuitos lógicos) para los autómatas, matemático (determinación de lo que puede o no computarse, y del grado de complejidad de las computaciones) para la teoría de la computación, lingüísticos (estudio científico del lenguaje natural) para la teoría de lenguajes. El paso del tiempo ha ido haciendo emerger nuevas y múltiples relaciones entre dichas áreas y de ellas con la gran mayoría de los desarrollos técnicos de la informática, por lo que su cabal conocimiento añade al dividendo reconocido de los buenos fundamentos la promesa de su permanente rentabilidad. Hemos intentado dotar al libro de determinadas características, que a continuación resumiremos. En primer lugar, dos propiedades dignas de mención son por un lado su carácter básico y su carácter señalizador. El adjetivo *básico* debe tomarse en su acepción de *fundamental* y no de *elemental*, ya que estamos presuponiendo en poder del lector los conocimientos equivalentes a una formación introductoria sólida sobre generalidades y estructura de ordenadores y sobre programación, más un lenguaje de alto nivel.

Por el término *señalizador* queremos aludir a la incorporación de numerosos apartados que introducen (a menudo en forma pormenorizada) a cuestiones directa o indirectamente relacionadas con técnicas informáticas muy actuales o con caminos científicos que se están abriendo. En pocas palabras, buscamos comprometer - en la medida de lo factible - el interés del lector, y lo que queremos significar con este objetivo se comprenderá mejor cuando entremos dentro de unos momentos a relacionar algunos aspectos de los contenidos del libro.

Nuestra esperanza, sin embargo, reside en que su propiedad más notoria sea la pedagógica, pues no en balde este libro ha tenido una vida anterior en la que a lo largo de tres ediciones consecutivas y varias reimpresiones ha servido de texto durante ocho o nueve años en la Escuela Técnica Superior de Ingenieros de Telecomunicación de Madrid, donde ambos autores somos profesores. Era el segundo de una obra en dos volúmenes, titulada genéricamente Fundamentos de los Ordenadores y editada por dicha Escuela.

Sobre ese material el presente libro contiene aproximadamente un cuarenta por ciento de cuestiones nuevas, lo que ha llevado a eliminar bastantes de las anteriores y a remodelar el sesenta por ciento restante. Aunque sus cualidades didácticas tendrán que ser juzgadas por los lectores, en lo que a nosotros concierne hemos puesto el máximo cuidado en estructurar y redactar los textos con la mayor claridad de que hemos sido capaces, y utilizando tanto cuanto hemos creído conveniente la ayuda de los ejemplos. Al formato también le hemos dado importancia. Y es así que todos los temas se organizan idénticamente por capítulos,

y todos los capítulos, menos el primero, poseen la misma estructura, pues cualquiera que sea su desarrollo terminan con un apartado de Resumen, un apartado llamado de Notas Histórica y Bibliográfica y (siempre que procede) un apartado de Ejercicios. El primer capítulo de cada tema es especial, ostenta el nombre de Ideas Generales y la misión de exponer un planteamiento panorámico del conjunto de los capítulos y de sus relaciones con los otros tres temas. Al final del libro una bibliografía ordenada alfabéticamente recoge exclusivamente las referencias citadas en las Notas de todos los capítulos, por lo que huelga decir que, dado el enfoque primordialmente orientador de las Notas, la bibliografía no pretende alcanzar ninguna meta de exhaustividad científica.

Por lo general, de los contenidos de un libro poco cabe decir en un prólogo, ya que su índice normalmente excusa de tal esfuerzo, salvo que se entienda que no ha de resultar ocioso captar la atención del lector acerca de diversas singularidades. Haremos de éstas un bosquejo rápido.

Refiriéndonos a la primera parte, es del común saber la naturaleza formativa de la lógica formal para el razonamiento, pero en el campo de la informática destaca además su naturaleza fundamentadora para numerosas aplicaciones de inteligencia artificial, de programación y de diseño de circuitos. Este segundo aspecto es el que hemos enfatizado en el tema de "Lógica", buscando, por ejemplo, culminar la exposición de las lógicas proposicional y de predicados de primer orden en el desarrollo de sistemas inferenciales, con la vista puesta en los sistemas expertos (dominio aplicado y en auge de la inteligencia artificial), a los que se dedica asimismo un amplio capítulo. Otro nos habla de Otras Lógicas que se prefiguran en el horizonte como prometedoras bases para los mencionados dominios aplicativos: ampliaciones de la lógica de predicados (lógica modal y temporal), lógicas multivaloradas y, en especial, un extenso desarrollo de la lógica borrosa. Una cuestión clásica como los circuitos combinacionales tiene aquí su capítulo, lo mismo que en el tema de "Autómatas" lo tienen los circuitos lógicos secuenciales.

Con respecto a la parte llamada "Autómatas", que versa sobre un área de expresión típicamente matemática, hemos suavizado mucho su formalismo descriptivo (a fin de cuentas, los autómatas de los que nos ocupamos son construcciones formales). No solamente bastantes teoremas se enuncian y se explican sin demostrarlos, también se usan ejemplos prácticos sencillos, entre los que, por ilustrar, citamos el detector de paridad, el sumador, el problema del castillo encantado, el contador, el reconocedor de una cadena de símbolos, etc. Como una muestra, entre otras, de nuestro declarado objetivo de interconectar los cuatro temas del libro está el capítulo dedicado a Autómatas Reconocedores y Lenguajes Regulares, tradicional por otro lado en el área de estudio de los autómatas.

Menos tradicional en un libro sobre principios básicos resulta aquel otro capítulo destinado a introducir Otros Autómatas, a saber: las redes de Petri, que han adquirido considerable predicamento en el ámbito del diseño de sistemas concurrentes, los autómatas estocásticos, los autómatas de aprendizaje y los autómatas borrosos.

En "Algoritmos" se establecen con cuidado definiciones de los conceptos de algoritmo, programa y máquina y sus profundas interrelaciones. Se ha tomado un esmero especial para que los lectores comprendan bien el análisis y diseño de máquinas de Turing, así como el significado de piedra angular de esta máquina en la teoría informática, en conexión con las cuestiones de la computabilidad. Por último, un capítulo sobre Complejidad intenta exponer las nociones esenciales que, en nuestra opinión, todo informático debe poseer sobre un campo teórico y práctico de gran desarrollo en los últimos años. La complejidad algorítmica se aplica en dominios tan diversos como la robótica, el diseño de circuitos integrados a muy grande escala y el diseño de estructuras de datos eficientes.

Los lenguajes formales, en general, y con un mayor detalle los lenguajes libres de contexto y las relaciones entre los diversos tipos de lenguaje y las estructuras de máquina capaces de reconocerlos, constituyen el contenido de los cuatro primeros capítulos de la última parte del libro. Es ésta un área obligada si se quiere comprender algo sobre la esencia de los lenguajes para ordenador. Precisamente, el extenso capítulo 5, *Aplicaciones a los lenguajes de programación*, intenta salvar un poco la brecha explicativa con la que nos encontramos cotidianamente entre la teoría de los lenguajes formales y su materialización en las herramientas que todo el mundo utiliza para construir el software. De las diversas cuestiones tratadas en ese capítulo, tal vez puedan destacarse por su interés sistemático las referidas a sintaxis, semántica y procesadores de lenguaje. Como sección incitadora de novedades y de futuro es de subrayar el apartado acerca de los lenguajes de la programación declarativa (programación funcional, programación lógica, etc.).

Nuestros alumnos de la rama de Informática de la Escuela se encontrarán a partir de este momento con un texto completamente nuevo. Comoquiera que hace dos años no se reimprimía aquél del que éste es una reencarnación, ellos se van a llevar una alegría, acaso un tanto atemperada por causa de los gajes del oficio de tener que estudiárselo. Al mismo tiempo, estamos razonablemente seguros de que también se alegrarán (con nosotros) de saber que desde ahora podrán compartir esta fuente de conocimiento con otros estudiantes y con la vasta comunidad de profesionales informáticos en el ámbito de la lengua castellana. Que así sea.

Los autores
Febrero, 1987

Anotamos aquí una especie de descripción casi notarial y ordenada (tema por tema) de las novedades reseñables de este libro con respecto al libro anterior. En cualquier caso, todos los capítulos aportan modificaciones menores (erratas, estilo, alguna aclaración adicional, referencias cruzadas, nueva bibliografía), que han supuesto finalmente más trabajo del que podíamos imaginar.

- *Tema "Lógica"*

Tanto el capítulo 2 (lógica de proposiciones) como el 4 (lógica de predicados) se han modificado sustancialmente. Concretamente, la semántica queda ahora definida de manera más rigurosa, porque la experiencia docente nos ha enseñado que un mayor grado de formalización en este aspecto no perjudica a la comprensión de los conceptos (si se insiste con ejemplos adecuados) y deja en mejor disposición al lector para entender sus aplicaciones (por ejemplo, en el estudio de las bases de datos deductivas).

El capítulo 3 (circuitos lógicos combinacionales) es el más ligado a la tecnología electrónica y, sin embargo, apenas ha sido necesario retocarlo (salvo, naturalmente, en las orientaciones bibliográficas, que se han actualizado). Esto tiene fácil explicación: sus contenidos, básicos, no han cambiado. Lo mismo puede decirse del capítulo 3 del tema "Autómatas" (circuitos secuenciales).

Tampoco el capítulo 5, dedicado a "otras lógicas", requería grandes cambios. El crecimiento de las aplicaciones de la lógica borrosa en los últimos años demuestra que nuestra decisión de hacer un especial énfasis en ella era adecuada. Anecdóticamente, nos congratula que nuestro ejemplo del "aprendiz de conductor", introducido con la intención de motivar al lector, se utiliza con la misma intención y en términos muy parecidos en el editorial del primer número de una revista especializada de reciente aparición (IEEE Transactions on Fuzzy Systems, Feb. 1993).

Al capítulo 6 se le ha dotado de una introducción más amplia y actualizada sobre la inteligencia artificial y la ingeniería del conocimiento. También se han añadido algunas observaciones sobre el uso de la lógica de predicados.

- *Tema "Autómatas"*

El capítulo 5 se ha ampliado con autómatas celulares (cuyo interés ha aumentado en los últimos años por su relación con los fractales y sus aplicaciones en procesamiento de imágenes) y, sobre todo, con una extensa introducción a los sistemas de aprendizaje, acompañada de numerosas orientaciones bibliográficas.

- *Tema "Algoritmos"*

El capítulo 3 sobre programación estructurada es una refundición abreviada de los capítulos 3 y 4 del anterior libro, en el que se ha añadido una sección sobre programación con Pascal y C.

El capítulo 4, que hemos titulado "Evolución de la programación" es absolutamente nuevo y -para resumirlo de forma muy esquemática- trata más que nada de la programación, el análisis y el diseño orientado a objetos. Consideramos a este estilo de programación un paradigma de futuro muy prometedor, que se extiende a todo un conjunto de nuevas prácticas informáticas conocido como "tecnología de objetos". Por esta razón, se ha hecho un esfuerzo en escribir una sección de "Notas histórica y bibliográfica" especialmente amplia para lo que son

los estándares de este libro. *Queremos rendir un tributo de amistad y gratitud a la insuperable colaboración del profesor Manuel Alfonseca Moreno en la redacción de este capítulo y en la refundición del capítulo 3.*

- *Tema "Lenguajes"*

En principio, el capítulo 5, dedicado a las "aplicaciones a los lenguajes de programación", tampoco requería muchos cambios. Desde que se publicó el primer libro, se ha progresado en cuanto al procesamiento de lenguajes, pero más bien en las técnicas ligadas a la arquitectura hardware de las máquinas (optimización en procesadores RISC y vectoriales, etc), no en lo relativo a los principios que aquí se tratan. Creemos que confirma esta aserción el hecho de que el clásico libro de Aho, Sethi y Ullman, de 1986, sigue siendo el libro de texto más utilizado. Sin embargo, nos ha parecido conveniente, puesto que ya se hablaba de lenguajes declarativos, ampliar la sección del procesamiento de lenguajes con una breve descripción del funcionamiento de los intérpretes de Prolog. También se ha añadido una introducción a las herramientas para la generación de procesadores.

*Los autores
Septiembre, 1995*

Parte I

Lógica

Fundamentos de informática

1



Ideas generales

1. Razonamientos y lógica formal

La lógica formal se ocupa de las *inferencias*, y éstas son *razonamientos formalmente válidos*. Empecemos por precisar esta declaración, explicando lo que entendemos por "razonamiento", por "válido" y por "formal".

Un *razonamiento* es un tipo de pensamiento que consiste en obtener una conclusión a partir de unas premisas. Aunque raras veces se explicita la distinción, el término "razonamiento" tiene dos referentes: uno *procesal* (la actividad del agente que razona) y otro *funcional* (la asociación entre las premisas y la conclusión).¹

¹ La definición de "razonamiento" que puede encontrarse en cualquier diccionario recoge implícitamente estas dos acepciones: "*acción y efecto* de razonar". El "efecto" guarda relación con lo funcional, o estático, y la "acción" con lo procesal, o dinámico. El adjetivo "procesal" no es frecuente en informática, pero su adecuación parece clara: "relativo a los procesos". El reconocimiento de la existencia de tres tipos de *modelos* (*estructurales, funcionales y procesales*) resulta de gran utilidad para el estudio de los ordenadores y los sistemas informáticos (Fernández, 1994).

La lógica se ocupa de los razonamientos en el sentido funcional. De los aspectos procesales se ocupa la psicología, en el caso de que el "agente" sea humano. Pero si el agente es un artefacto (que, con la tecnología actual, es lo mismo que decir un ordenador) entonces es algo propio de la informática (o de una rama de ésta conocida como "inteligencia artificial", sobre la que hablaremos algo más adelante y en el capítulo 6).

El calificativo "*válido*", aplicado a un razonamiento, es sinónimo de "*deductivo*", y se aplica a aquel razonamiento en el cual se cumple que si las premisas son verdaderas entonces podemos asegurar que la conclusión también lo es. Este concepto reviste gran importancia. En el siguiente capítulo volveremos sobre él, pero vamos a ilustrarlo ya con un ejemplo:

Premisa 1: Todos los libros sobre informática son terriblemente aburridos.

Premisa 2: Este es un libro sobre informática.

Conclusión: Este libro es terriblemente aburrido.

Sobre la verdad o la falsedad de estas dos premisas y de la conclusión pueden darse todas las combinaciones posibles, salvo una. En efecto, se puede "poner en duda", o, mejor dicho, "negar"² alguna de la premisas, o ambas, y considerar que la conclusión es falsa. Pero también se puede negar cualquiera de las premisas y, sin embargo, considerar verdadera a la conclusión (las premisas *no son necesarias* para la conclusión). Lo que de ninguna manera suponemos posible es que, razonando correctamente, alguien esté de acuerdo con ambas premisas y no con la conclusión (las premisas *son suficientes* para la conclusión).

Por último, "*formal*" quiere decir que se atiende exclusivamente a la forma, no al contenido del razonamiento. El ejemplo anterior y el clásico:

Premisa 1: Todos los hombres son mortales.

Premisa 2: Sócrates es un hombre.

Conclusión: Sócrates es mortal.

no es que sean "similares". Es que, formalmente, son el mismo razonamiento. Ambos obedecen al esquema formal:

Premisa 1: Todos los individuos u objetos que tienen la propiedad p también tienen la propiedad q .

² La lógica que, en principio, vamos a considerar es "binaria", y en ella no se puede "dudar": las afirmaciones son o bien verdaderas o bien falsas. En el capítulo 5 estudiaremos otros tipos de lógica que se adaptan mejor a los modos de razonamiento humanos. En particular, esperamos, obviamente, que el lector dude, al menos, de la primera premisa y de la conclusión.

Premisa 2: El individuo u objeto x tiene la propiedad p .

Conclusión: El individuo u objeto x tiene la propiedad q .

2. Lógica e informática

Así pues, la lógica es una herramienta para el análisis de los razonamientos o argumentaciones generados por la mente humana. ¿Y qué relación puede haber entre la lógica y la informática? Pues bien, podemos mencionar, al menos, dos puntos de contacto:

(a) *En aplicaciones "inteligentes"*. Ya lo hemos mencionado más arriba, al señalar que el "agente" que razona puede ser una persona o una máquina. Extendámonos un poco más sobre esta idea. Los ordenadores son máquinas diseñadas para mecanizar trabajos intelectuales. Normalmente, esos trabajos son los relacionados con tareas sencillas y rutinarias: cálculos basados en operaciones aritméticas (que el hombre aprende de memoria y aplica sin necesidad de razonar), búsqueda de datos (por simple comparación o emparejamiento con una clave dada), clasificación (ordenación de datos basada en un criterio elemental), etc. Para estas aplicaciones no hace falta aplicar la lógica (en el sentido de ciencia; naturalmente que todo informático, como todo ser racional, utiliza razonamientos lógicos de manera informal). Si pretendemos mecanizar tareas más complejas (inducción, deducción, etc.) entramos en el campo de la informática llamado *inteligencia artificial*³. Ahora se trata de conseguir que la máquina sea capaz de hacer, precisamente, esos razonamientos que, de manera informal, realiza el hombre. Y para ello es preciso analizarlos, definirlos con precisión, en una palabra, formalizarlos. Y eso es precisamente lo que hace la lógica (formal).

(b) *En programación*. Hace ya años que se desencadenó la que se llamó "crisis del software": programas cada vez más complejos, menos fiables y de mantenimiento más difícil. Se han propuesto y se utilizan diversas metodologías para la construcción de programas, basadas en principios teóricos que se tratarán en el tema "Algoritmos". Pero un enfoque complementario de tales metodologías es el de buscar una programación "declarativa" en lugar de "imperativa". Esto quiere decir que se trataría de buscar lenguajes de programación tales que los programas no sean una secuencia de instrucciones que le digan al ordenador, paso a paso, cómo hay que resolver el problema, sino una especificación de qué es lo que se pretende resolver, y que sea el propio ordenador quien determine las acciones

³ En realidad, la inteligencia artificial es un campo interdisciplinar, que interesa también a la psicología y a la epistemología; aquí hablamos de "aplicaciones inteligentes de los ordenadores", y, por tanto, nos referimos a la vertiente más ingenieril.

necesarias para ello. Pues bien, la lógica puede verse, precisamente, como un lenguaje de especificación mediante el cual podemos plantear los problemas de manera rigurosa. En el tema "Lenguajes" comentaremos los principios de la programación lógica. Por otra parte, la tarea de programación se complica cuando se hace preciso considerar procesos concurrentes e intercomunicantes (por ejemplo, en los sistemas operativos de multiprogramación, en las aplicaciones de "tiempo real", o en los sistemas distribuidos); ciertas extensiones de la "lógica clásica", como las lógicas modal y temporal que veremos en el capítulo 5, son herramientas matemáticas adecuadas para estos casos.

3. Lógica y circuitos de conmutación

Desde el punto de vista estrictamente electrónico, el soporte tecnológico principal de los ordenadores lo constituyen los circuitos de conmutación, o "circuitos lógicos". Esta última denominación es, desde luego, discutible. Porque los circuitos electrónicos básicos, con la tecnología actual, no permiten realizar directamente las operaciones inferenciales que caracterizan a la lógica. (Aunque indirectamente sí, ya que podemos, por ejemplo, programar a un ordenador para que las haga, e, incluso, podemos pensar en nuevas estructuras de ordenador cuyas operaciones elementales no sean ya las instrucciones de máquina de los ordenadores actuales, sino inferencias lógicas. Este es un campo de la arquitectura de ordenadores sometido a intensa investigación). Hay, no obstante, motivos para llamar "lógicos" a tales circuitos. Motivos que pueden reducirse a una consideración histórico-matemática: el hecho de que las formas elementales de la lógica y los circuitos de conmutación tienen un modelo matemático común, el álgebra de Boole binaria, introducida en el siglo pasado para formalizar la lógica y aplicada en éste como herramienta para el análisis y diseño de circuitos. Dada la importancia de los circuitos lógicos como componentes, y aunque no tengan una relación demasiado estrecha con el resto de las aplicaciones informáticas de la lógica, dedicaremos un capítulo de este Tema a su estudio.

4. Lógica, lenguajes y autómatas

Un lenguaje es un sistema de símbolos y de convenios que se utiliza para la comunicación, sea ésta entre personas, entre personas y máquinas o entre máquinas. El estudio matemático de los lenguajes es uno de los pilares de la informática, y a él vamos a dedicar parte del tema "Autómatas" y todo el último Tema del libro. Pero la lógica formal también puede considerarse como un lenguaje (aunque éste es un asunto debatido filosóficamente), "el mejor hecho de los lenguajes", como dice Ferrater Mora.

Por otra parte, existe una correspondencia muy estrecha, como veremos en los Temas correspondientes, entre los autómatas y los lenguajes: a cada tipo de autómata corresponde un tipo de lenguaje, y viceversa. La máquina de Turing es una máquina lógica, un autómata de un tipo especial, que ha permitido formalizar el concepto de algoritmo y ha proporcionado una de las aproximaciones teóricas al campo de la computabilidad, como se verá en el tema "Algoritmos".

Puesto que la terminología asociada a la teoría de lenguajes va a utilizarse en todas las partes de este libro, parece conveniente presentar ya las definiciones y los conceptos básicos.

5. Lenguajes formales: definiciones y conceptos básicos

5.1. Alfabeto, cadenas y lenguaje universal

Llamaremos *alfabeto* a un conjunto finito, no vacío, de símbolos.

Una *cadena* (o palabra, o expresión, o secuencia finita) es una secuencia ordenada, finita, con o sin repetición, de los símbolos de un alfabeto. De una manera general, utilizaremos las letras x, y, z para representar cadenas construidas con cualquier alfabeto (que, por supuesto, no puede contener esos símbolos). A veces interesa definir la *cadena vacía* como aquella que no contiene ningún símbolo. Para designarla utilizaremos el símbolo especial " λ " (en el supuesto, naturalmente, de que este símbolo no forma parte del alfabeto con el que estamos construyendo las cadenas).

Sobre el conjunto de cadenas construidas con un determinado alfabeto puede definirse una función total, llamada longitud, y abreviada " lg ", que asigna a cada cadena el número de símbolos de que consta. Por ejemplo, si el alfabeto es $\{0,1\}$, entonces $lg(0) = 1$, $lg(1) = 1$, $lg(00) = 2$, $lg(01) = 2$, etc. Por definición, $lg(\lambda) = 0$.

Si A es un alfabeto, llamaremos A^n al conjunto de todas las cadenas de longitud n . Por ejemplo, si $A = \{a,b\}$, entonces,

$$A^0 = \{\lambda\}; A^1 = \{a, b\}; A^2 = \{aa, ab, ba, bb\}, \text{ etc.}$$

Llamaremos lenguaje universal sobre un alfabeto A , y lo representaremos por A^* , al conjunto infinito

$$A^* = A^0 \cup A^1 \cup A^2 \cup \dots = \bigcup_{i=0}^{i=\infty} A^i$$

Es decir, A^* es el conjunto de todas las cadenas (incluida λ) que pueden formarse a partir de A .

5.2. Concatenación

La *concatenación* de dos cadenas, $x, y \in A^*$, es una ley de composición interna sobre A^* , es decir, es una aplicación $A^* \times A^* \rightarrow A^*$, que consiste en formar la cadena xy poniendo x delante de y .

Por ejemplo, si $A = \{a, b, c\}$ y consideramos las cadenas $x = ab$, $y = bba$, entonces $xy = abbba$, y $yx = bbaab$.

La representación de la concatenación de una cadena consigo misma puede abreviarse mediante notación exponencial: $x^0 = \lambda$; $x^1 = x$; $x^2 = xx$; $x^3 = xxx$, etc.

(Obsérvese, aunque sea a título anecdótico, que la longitud cumple respecto a la concatenación las mismas propiedades formales que el logaritmo respecto a la multiplicación: $\lg(xy) = \lg(x) + \lg(y)$; $\lg(x^n) = n \cdot \lg(x)$).

La operación de concatenación satisface las siguientes propiedades:

- a) Es asociativa: $x(yz) = (xy)z = xyz$
- b) En general, no es conmutativa: $xy \neq yx$. (La conmutatividad sólo se da en el caso particular de que A conste de un solo elemento).
- c) Tiene un elemento neutro, λ : $\lambda x = x\lambda = x$.
- d) Ninguna cadena (salvo λ) tiene inverso: dada $x \in A$ ($x \neq \lambda$), no existe ningún $y \in A$ tal que $xy = \lambda$.

Por consiguiente, $\langle A^*, >$, es decir, A^* con la operación de concatenación, es una estructura algebraica de tipo monoide, a la que se llama *monoide libre generado por A*.

También es posible prescindir de la cadena vacía y trabajar con $A^+ = A^* - \{\lambda\}$ en lugar de con A^* . Algebraicamente, la diferencia estaría en que, al no existir elemento neutro, en lugar de hablar de monoide tendríamos que hablar de semigrupo.

5.3. Lenguaje y metalenguaje

Dado un alfabeto A , todo subconjunto de A^* se llama *lenguaje* sobre A . Esta definición es rigurosamente válida, pero de poca utilidad. Porque en la aplicación del concepto de lenguaje (ya sea a asuntos prácticos, como en los lenguajes de programación, o teóricos, como en la lógica formal) lo que interesa es poder resolver dos problemas duales: dado un lenguaje, *generar* cadenas que pertenezcan a ese lenguaje, y dada una cadena, *reconocer* si pertenece o no a un determinado lenguaje. Si el lenguaje en cuestión es finito (es decir, consta de un número finito de cadenas), la solución de ambos problemas es inmediata, puesto que el lenguaje se describe mediante la simple enumeración de sus cadenas. Pero lo

normal es que el lenguaje sea infinito, y entonces tendremos que dar una *descripción finita* para poder resolver esos problemas.

Y esto nos lleva al concepto de metalenguaje, porque dicha descripción finita tendrá que hacerse utilizando un sistema de símbolos que no pueden ser los mismos del lenguaje que trata de describir. Por tanto, tal descripción habrá de hacerse en otro lenguaje, que, como sirve para describir al primero (*lenguaje objeto*), se llama *metalenguaje*. Por ejemplo, en el lenguaje del álgebra elemental, escribimos expresiones como " $x + 7 = 17$ "; cuando le explicamos a alguien lo que significan expresiones de ese tipo, estamos utilizando el español como metalenguaje para describir el lenguaje del álgebra. Cuando hablamos de propiedades del lenguaje natural, español en nuestro caso, solemos utilizarlo indistintamente como lenguaje objeto o como metalenguaje. Por ejemplo, si decimos "el ordenador no es más que una máquina", hemos construido una frase del lenguaje en la que se *usa*, entre otras, la palabra "ordenador". Y si decimos "ordenador" es un vocablo de cuatro sílabas" también hacemos una frase, pero en ella, la palabra "ordenador" no se *usa*, se *menciona*. En realidad, estamos acudiendo al español como metalenguaje para describir una cierta propiedad del propio lenguaje. Y puede haber una jerarquía de lenguajes, es decir, varios niveles de descripción. Así, una frase del "metametalenguaje" sería: «"ordenador" es un vocablo de cuatro sílabas» es un enunciado verdadero".

La descripción de un lenguaje artificial suele hacerse mediante unas reglas que permiten generar cadenas pertenecientes al lenguaje ("cadenas o expresiones válidas", o "sentencias", o "frases"). Son las "reglas de formación" que veremos en este Tema, o las "reglas de escritura" que definiremos en el tema "Lenguajes". También, para ciertos tipos de lenguajes es posible dar descripciones algebraicas, como veremos en el tema "Autómatas" para los lenguajes llamados regulares.

5.4. Sintaxis, semántica y pragmática

En la lingüística hay tres campos: la sintaxis, la semántica y la pragmática. La sintaxis se ocupa estrictamente de los símbolos y de la manera de combinarlos para obtener sentencias del lenguaje. La semántica estudia los símbolos y las sentencias en relación con los objetos que designan. La pragmática, finalmente, trataría (en realidad, es el nivel menos desarrollado, y roza ya con los campos de la psicología y la sociología) de las relaciones entre los símbolos y los sujetos que los usan.

Un ejemplo de enunciado perteneciente a la sintaxis es:

'algoritmo' es un sustantivo.

Un ejemplo de enunciado perteneciente a la semántica es:

no es cierto que 'algoritmo' derive de 'αλγοζ' ('dolor') y 'αριθμοζ' ('número') y signifique 'dolor producido por los números'.

Finalmente,

en el Tema 3 daremos la etimología correcta del vocablo 'algoritmo'

es un enunciado perteneciente a la pragmática.

En el primer caso se ha usado 'algoritmo' para relacionarlo con otra expresión ('sustantivo'), en el segundo, para relacionarlo con el objeto que designa (en este caso, más bien con el objeto que *no* designa), y en el último, para relacionarlo con quienes lo usan (nosotros).

6. Resumen

La lógica formal es uno de los fundamentos teóricos de la informática, que, además, tiene importancia práctica en las aplicaciones llamadas de "inteligencia artificial", en los lenguajes de programación declarativos, en la formalización de lenguajes de programación en general, y en la programación concurrente.

Los circuitos lógicos, sin otra relación con la lógica formal que la existencia de un modelo matemático común, son los componentes tecnológicos básicos de los ordenadores actuales.

Los conceptos de alfabeto, cadenas, concatenación, lenguaje, sintaxis y semántica, definidos en este capítulo, se utilizarán en el resto de este Tema, y también en otros Temas del libro.

Fundamentos de informática

2



2

Lógica de proposiciones

1. Introducción

1.1. Variables proposicionales y sentencias

Los lógicos clásicos distinguían entre *juicio* (el acto mental de pensar o concebir algún hecho elemental), *proposición* (lo pensado o concebido en ese acto mental) y *razonamiento* (combinación de unas proposiciones llamadas *premisas* con otra llamada *conclusión*).

La noción de proposición se formaliza mediante el concepto de *variable proposicional*. Las variables proposicionales representan proposiciones que han de corresponder a *enunciados declarativos*, es decir, frases expresadas en el modo gramatical indicativo.¹ Para escribir variables proposicionales utilizaremos las letras *p, q, r, s, t...*, eventualmente con subíndices.

¹La formalización de modos subjuntivo, condicional e imperativo exige otros tipos de lógica.

Una *sentencia* representa a un enunciado compuesto por proposiciones (enunciados elementales) y determinadas operaciones primitivas ("y", "o", "si...entonces...", etc.), operaciones que se formalizan mediante las llamadas *conectivas*. Como veremos enseguida, las premisas y la conclusión de un razonamiento, y el razonamiento mismo, se formalizan con sentencias.

1.2. Conectivas

La conectiva unaria (o monádica) de *negación*, "no", cuyo efecto es negar lo que dice el enunciado que le sigue, se representa mediante el símbolo " \neg " antepuesto a la variable proposicional o a la sentencia correspondiente. Así, si con " p " estamos formalizando "la nieve es blanca", " $\neg p$ " representaría "la nieve no es blanca". (En otros libros se utilizan otros símbolos: " $\sim p$ ", o " \bar{p} ", o " p^* ").

Las conectivas binarias (o diádicas) son las que enlazan entre sí a dos variables proposicionales o a dos sentencias. Las más utilizadas son:

- la *conjunción*, con símbolo " \wedge " (en otros libros, "&" o "."), que representa a la conjunción ("y") del lenguaje natural;
- la *disyunción*, con símbolo " \vee " (en otros libros, "|" o "+"), que representa a la disyunción ("o") del lenguaje natural (en sentido *incluyente*: " $p \vee q$ " significa "o bien p , o bien q , o bien ambos").
- el *condicional*, con símbolo " \rightarrow " (en otros libros, " \Rightarrow " o " \supset "), que representa el "si...entonces...". De las dos variables enlazadas por el condicional, la primera (la de la izquierda) se llama *antecedente* y la segunda, *consecuente*.²
- el *bicondicional*, con símbolo " \leftrightarrow " (en otros libros, " \Leftrightarrow " o " \equiv "), que representa el "si y sólo si".

Combinando de forma adecuada variables y conectivas se forman sentencias, o cadenas correctas en el lenguaje de la lógica de proposiciones.

1.3. Interpretaciones y evaluaciones binarias de variables proposicionales y de sentencias

Aunque en el apartado 3, dedicado a la semántica, definiremos con mayor rigor los conceptos de interpretación y de evaluación, vamos a introducirlos ya, limitándonos al caso particular (y más sencillo) de la *evaluación binaria*.

² El símbolo " \rightarrow ", que utilizaremos en adelante, coincide con el que se utiliza en matemáticas para representar una aplicación, del que también tendremos que servirnos ocasionalmente (de hecho, ya ha aparecido en el capítulo 1, apartado 5.2, al definir la operación de concatenación de cadenas). Por el contexto, quedará claro a qué nos estamos refiriendo en cada caso.

Hemos dicho que las variables proposicionales *representan* proposiciones (enunciados declarativos elementales), y las sentencias, en general, *representan* enunciados declarativos. Esta idea de "representación" se formaliza con una función que hace corresponder a cada variable proposicional y a cada sentencia un enunciado, y esta función se llama *interpretación*. Así, ¿qué representan las variables p y q ? Depende de la interpretación. Una interpretación, i_1 , podría ser, por ejemplo:

$$i_1(p) = \text{la nieve es blanca};$$

$$i_1(q) = \text{el carbón es negro}.$$

Otra interpretación, i_2 , podría ser:

$$i_2(p) = \text{la nieve es blanca};$$

$$i_2(q) = \text{el carbón es blanco}.$$

La interpretación de sentencias se obtiene teniendo en cuenta lo que representan las distintas conectivas, de modo que, dadas una sentencia y una interpretación de las variables proposicionales que intervienen en ella, resulta una interpretación única para esa sentencia. Así, a la sentencia $p \wedge q$ le corresponden, para cada una de las interpretaciones dadas más arriba, estas interpretaciones:

$$i_1(p \wedge q) = \text{la nieve es blanca y el carbón es negro};$$

$$i_2(p \wedge q) = \text{la nieve es blanca y el carbón es blanco}.$$

Una *evaluación binaria* consiste en asignar a cada una de las variables proposicionales uno de estos dos valores: "verdadero" (o "cierto") o "falso". En lo sucesivo, utilizaremos el símbolo "1" para la representación de "verdadero" y "0" para la de "falso". Obsérvese que la evaluación es siempre *relativa a una interpretación*. Con los ejemplos anteriores, y representando por e_1 y e_2 las evaluaciones relativas a las interpretaciones i_1 y i_2 respectivamente, tendremos:

$$e_1(p) = 1;$$

$$e_1(q) = 1;$$

$$e_2(p) = 1;$$

$$e_2(q) = 0;$$

Dada una evaluación de las variables proposicionales que intervienen en una sentencia, para poder dar una evaluación a la sentencia, es decir, para poder decir

si la sentencia es verdadera (valor 1) o falsa (valor 0), tenemos que dar un significado a las conectivas. El significado universalmente admitido es el que se resume en la siguiente tabla:

	p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
e_0	0	0	1	0	0	1	1
e_1	0	1	1	0	1	1	0
e_2	1	0	0	0	1	0	0
e_3	1	1	0	1	1	1	1

Cada una de las filas de la tabla corresponde a una evaluación de las variables proposicionales p y q . En el caso binario sólo hay cuatro evaluaciones distintas de dos variables proposicionales: ambas falsas ($i_0(p) = 0, i_0(q) = 0$), una verdadera y otra falsa ($e_1(p) = 0, e_1(q) = 1$; $e_2(p) = 1, e_2(q) = 0$) o ambas verdaderas ($e_3(p) = 1, e_3(q) = 1$). Para cada una de estas evaluaciones, la tabla muestra la evaluación de la sentencia formada mediante la negación de una variable proposicional (columna de $\neg p$) o mediante la unión de dos variables proposicionales con una conectiva binaria (columnas siguientes). Esta forma de dar significado a las sentencias que se forman con variables proposicionales y conectivas merece algunos comentarios.

- *Negación.* Parece natural que si una variable proposicional representa un hecho verdadero, su negación sea un hecho falso, y viceversa. Por ejemplo, si p representa "la nieve es blanca", y la evaluamos como verdadera, $\neg p$ representará "la nieve no es blanca", y se evaluará como falsa.
- *Conjunción.* También parece natural que la conjunción de dos hechos sólo sea un hecho verdadero si lo es cada uno de ellos separadamente: si $p \wedge q$ representa "la nieve es blanca y me apetece esquiar", $p \wedge q$ sólo es verdadero si, además de serlo p , verdaderamente me apetece esquiar.
- *Disyunción.* Como ya se ha dicho más arriba, la interpretación que se da a la disyunción de dos variables proposicionales es la inclusiva: basta con que una de ellas sea verdadera para que su disyunción también lo sea. Es decir, lo que representa $p \vee q$ es "o bien p , o bien q , o ambos". Por tanto, no sirve para representar enunciados como "o viaje en avión o viaje en tren (pero no en ambos)". (La representación de este enunciado, en el que se usa la disyunción exclusiva, sería, con las conectivas que tenemos: $(p \wedge \neg q) \vee (\neg p \wedge q)$, es decir, traduciendo al lenguaje natural: "una de dos: o bien viaje en avión (p) y entonces no viaje en tren, o bien viaje en tren (q), y entonces no viaje en avión").

- *Condicional*. Como puede verse en la tabla, la evaluación de dos variables enlazadas por el condicional es tal que la sentencia sólo se evalúa como falsa en el caso de que el antecedente sea verdadero y el consecuente falso. Este sentido del condicional requiere una explicación algo más detallada. Para ello, analicemos cada uno de los casos con algunos ejemplos:

a) $e(p)=1$, $e(q)=1$ (antecedente y consecuente verdaderos). Parece obvio que en tal caso el condicional ("si p entonces q ") ha de evaluarse como verdadero. Así,

'si como mucho entonces engordo'

es una sentencia que se evalúa como verdadera en el caso de que tanto el antecedente ("como mucho") como el consecuente ("engordo") sean verdaderos. Pero piénsese que también han de evaluarse como verdaderos condicionales en los que no existe una relación causal entre el antecedente y el consecuente. Por ejemplo:

'si la nieve es blanca entonces el carbón es negro'

ha de evaluarse como verdadero. Por esta razón, aunque a veces al condicional se le llame *implicación material*, no hay que confundirlo con la *implicación lógica* ni la *implicación estricta*:

'la nieve es blanca implica que el carbón es negro'

es una implicación falsa, tanto en el sentido lógico como en el estricto. En el apartado 5.2 de este mismo capítulo definiremos la implicación lógica; la implicación estricta se estudia en la lógica modal, como veremos en el capítulo 5 (apartado 2.4).

b) $e(p)=1$, $e(q)=0$. En este caso, parece natural decir que el condicional $p \rightarrow q$ es falso. En efecto, decir "si p entonces q " es equivalente a decir "si p es verdadero entonces q es verdadero", o, lo que es igual, " p es condición suficiente (aunque no necesaria) de q "; por tanto, el hecho de que p sea verdadero y, sin embargo, q sea falso viene a refutar la sentencia $p \rightarrow q$, es decir, a hacerla falsa.

c) $e(p)=0$, $e(q)=1$. El sentido común nos indica que en este caso el condicional $p \rightarrow q$ no es ni verdadero ni falso: ¿qué sentido tiene preguntarse por la verdad o la falsedad de un condicional cuando la condición expresada por su antecedente no se cumple? Pero esta respuesta del sentido común no nos sirve, porque estamos trabajando con una *lógica binaria*, y todo tiene que evaluarse bien como verdadero bien como falso. Si una sentencia no es falsa entonces es verdadera, y viceversa. Ahora bien, en el

caso que nos ocupa podemos asegurar que el condicional no es falso. No lo es porque, como hemos dicho más arriba, " $p \rightarrow q$ " es igual que afirmar que p es condición suficiente *pero no necesaria* de q , es decir, que no es la única condición posible, por lo que perfectamente puede darse el caso de que q sea verdadero siendo p falso. Es decir, la falsedad del antecedente no hace falso al condicional. Y si no lo hace falso, lo hace verdadero. Por ejemplo: "si corro, entonces me canso". ¿Qué ocurre si se dan los hechos de que no corro y, sin embargo, me canso? Ello no invalida la sentencia, porque no se dice que no pueda haber otras causas que me producen cansancio.

d) $e(p)=0$, $e(q)=0$. La situación es algo parecida a la del caso anterior: la condición no se da (es falsa), por lo que q puede ser tan verdadero como falso, y el condicional, al no ser falso, será verdadero. Obsérvese, anecdóticamente, que no es raro encontrar este uso del condicional en el lenguaje coloquial, como queriendo señalar que, ante un dislate, cualquier otro está justificado: "si Fulano es demócrata yo soy el emperador de Asiria".

Este significado del condicional, aunque hoy está universalmente admitido, ha ocasionado numerosos problemas en lógica, y ha sido una fuente de paradojas.³ Por ejemplo, la sentencia " $p \rightarrow (q \rightarrow p)$ " tiene siempre la evaluación "verdadera", sea cual sea la evaluación de las variables p y q . En efecto, podemos construir la siguiente tabla para ver, en dos pasos, las cuatro evaluaciones posibles:

p	q	$q \rightarrow p$	$p \rightarrow (q \rightarrow p)$
0	0	1	1
0	1	0	1
1	0	1	1
1	1	1	1

Esto es lo que se llama una *tautología*: una sentencia que siempre es verdadera sean cuales sean las interpretaciones de las variables proposicionales que intervienen en ella. (Obsérvese que interpretaciones hay infinitas, pero evaluaciones diferentes sólo cuatro). A primera vista, este resultado puede parecer paradójico, porque, considerado como la formalización de un razonamiento en el que " p " fuese la premisa y " $q \rightarrow p$ " la conclusión, viene a decir que si un enunciado (p)

³ Incluso hoy, en ingeniería y en informática, donde el condicional se utiliza, entre otras cosas, para formalizar "reglas" (capítulo 6, apartado 2.3), persisten concepciones equivocadas (y hasta aparecen publicadas, véase, por ejemplo, Mendel, 1995) debidas a la identificación (por carencia de una sólida base de conocimientos sobre lógica) entre "condicional" e "implicación lógica".

es verdadero, todo condicional en el que ese enunciado sea el consecuente es verdadero, independientemente de que el antecedente sea verdadero o falso. Por ejemplo, de un enunciado como "tengo frío" se deduce que "si salgo al campo entonces tengo frío" siempre es verdadero, independientemente de que "salgo al campo" sea verdadero o falso. Ahora bien, teniendo en cuenta el sentido material del condicional, no es que "tengo frío" *implique* "si salgo al campo entonces tengo frío", sino que la segunda cosa *se sigue* de la primera (es verdadera cuando la primera lo es), y entonces la interpretación parece perfectamente razonable: si tengo frío, tengo frío, pase lo que pase.

- *Bicondicional.* $p \leftrightarrow q$ se evalúa como verdadero si las dos variables son verdaderas o las dos son falsas. En caso contrario (una verdadera y otra falsa) se evalúa como falso. Pueden hacerse consideraciones similares a las del condicional (e invitamos al lector a reflexionar sobre ello). De la misma manera que al condicional se le llama a veces "implicación" (material), al bicondicional se le conoce también como "equivalencia" (material, no lógica, ni estricta). Como ya hemos dicho, estas denominaciones ("implicación" y "equivalencia") deben eludirse cuando hablamos del condicional y del bicondicional, porque conducen a confusiones. Definiremos la equivalencia lógica en el apartado 3.5.

Una observación final: La tabla dada al principio de este apartado se puede escribir también utilizando dos sentencias cualesquiera, S_1 y S_2 , en lugar de las variables proposicionales p y q . Para una determinada evaluación de las variables que forman parte de ellas, S_1 y S_2 tendrán unas evaluaciones "0" o "1" que se calcularán de acuerdo con la misma tabla. Y todo lo que hemos explicado sobre el significado de las distintas conectivas aplicadas a variables es extensible al caso en que apliquen a sentencias.

1.4. Ejemplos de formalización e interpretación

Aunque más adelante presentaremos con mayor rigor estas primeras ideas, lo que hemos visto es ya suficiente para poder formalizar y analizar muchas frases, razonamientos y argumentaciones expresados en lenguaje natural.

En los ejemplos de este apartado partimos de frases del lenguaje natural, y lo primero que hacemos es formalizarlas. Para ello, creamos funciones inversas de la interpretación, es decir, inventamos variables proposicionales que representan a los enunciados elementales presentes en las frases originales, y construimos sentencias que representan a las frases. Una vez construida la sentencia, escribimos todas las evaluaciones de sus variables proposicionales; si hay n variables proposicionales, habrá 2^n evaluaciones binarias, que escribiremos en 2^n líneas. Después calculamos la evaluación de la sentencia para cada una de esas evaluaciones de variables proposicionales (lo cual puede exigir, para evitar errores, cálculos intermedios de las partes de la sentencia). El resultado es una tabla con 2^n filas, a la que llamaremos *tabla de verdad*. (Ya hemos construido una an-

teriormente, cuando comentábamos el significado el condicional). Lo interesante es que los resultados del análisis de la sentencia son generales y aplicables a cualquier otra interpretación. Veamos algunos ejemplos concretos.

Ejemplo 1.4.1. "Si corro entonces me canso". Ya hemos comentado antes este enunciado, que podemos formalizar, simplemente, como " $p \rightarrow q$ ". Si lo repetimos aquí es para invitar al lector a reflexionar sobre el hecho de que hay formas alternativas de decir lo mismo. Por ejemplo: " $\neg p \vee q$ " ("o bien no corro, o bien me canso, o ambos"), " $\neg(p \wedge \neg q)$ " ("no es el caso que corro y no me canso"), etc.:

p	q	$p \rightarrow q$	$\neg p$	$\neg q$	$\neg(p \vee q)$	$p \wedge \neg q$	$\neg(p \wedge \neg q)$
0	0	1	1	1	1	0	1
0	1	1	1	0	1	0	1
1	0	0	0	1	0	1	0
1	1	1	0	0	1	0	1

La primera de esas formas alternativas ($\neg p \vee q$) es una forma disyuntiva que, como veremos en los apartados 5.7 y 5.8, tiene importancia para ciertos procedimientos de inferencia automática.

Ejemplo 1.4.2. "Si tengo hambre o tengo sed, entonces voy al bar". Definiendo las variables proposicionales p , q , r para representar respectivamente "tengo hambre", "tengo sed" y "voy al bar", podemos formalizar la frase en cuestión mediante la sentencia:

$$p \vee q \rightarrow r$$

Como intervienen tres variables, hay $2^3 = 8$ evaluaciones binarias para el conjunto de las tres, y para cada una de ellas la sentencia tendrá una evaluación. Las ocho evaluaciones, que se calculan de acuerdo con lo dicho anteriormente, se resumen en esta tabla de verdad:

p	q	r	$p \vee q$	$p \vee q \rightarrow r$
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1

p	q	r	$p \vee q$	$p \vee q \rightarrow r$
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Como puede observarse, la sentencia es falsa cuando ocurre que "tengo hambre" o "tengo sed" (o ambos) son verdaderos y, sin embargo, "voy al bar" es falso. En cualquier otro caso, incluso cuando p y q son las dos falsas, la sentencia es verdadera (lo cual es razonable: la sentencia no dice que no se pueda ir al bar por otros motivos; lo diría si en lugar de un condicional se utilizase un bicondicional).

Ejemplo 1.4.3. Muchos razonamientos consisten en la obtención de una conclusión a partir de dos o más premisas: "si *Premisa1* y *Premisa2* y ... entonces *Conclusión*". Si las premisas y la conclusión pueden expresarse formalmente en lógica de proposiciones, el razonamiento es válido siempre y cuando la sentencia global $(P1 \wedge P2 \wedge \dots) \rightarrow C$ sea una tautología. En efecto, lo que queremos expresar con el razonamiento es justamente lo que expresa un condicional de esa forma: que en el caso de que las premisas sean verdaderas la conclusión también lo es. Y si hay algún caso en el cual todas las premisas son verdaderas y la conclusión falsa, entonces el razonamiento falla; en tal caso, la sentencia condicional dada se evalúa como falsa. Si nunca ocurre tal cosa, la sentencia es una tautología y el razonamiento es correcto.

Consideremos el siguiente razonamiento:

$P1$: 'si Bernardo se casa, Florinda se suicida'

$P2$: 'Florinda se suicida si (y solamente si) Bernardo no se hace monje'

C : 'si Bernardo se casa, no se hace monje'

Para formalizar este razonamiento, definamos tres variables proposicionales, c , s y m , con este significado (función de interpretación):

$i(c)$ = 'Bernardo se casa'

$i(s)$ = 'Florinda se suicida'

$i(m)$ = 'Bernardo se hace monje'

Los elementos del razonamiento se formalizarán con estas sentencias:

$$P1: c \rightarrow s$$

$$P2: s \leftrightarrow \neg m$$

$$C: c \rightarrow \neg m$$

y la formalización del razonamiento en su conjunto vendrá dada por la sentencia:

$$(c \rightarrow s) \wedge (s \leftrightarrow \neg m) \rightarrow (c \rightarrow \neg m)$$

Analicemos ahora las evaluaciones de esta sentencia para todas y cada una de las posibles evaluaciones de sus variables.

			$(c \rightarrow s) \wedge$		$(c \rightarrow s) \wedge (s \leftrightarrow \neg m)$
c	s	m	$c \rightarrow s$	$s \leftrightarrow \neg m$	$(s \leftrightarrow \neg m)$
			$c \rightarrow \neg m$	$\rightarrow (c \rightarrow \neg m)$	
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	0	1	0
1	1	0	1	1	1
1	1	1	1	0	0

Comprobamos así que se trata de una tautología, y, por consiguiente, el razonamiento es correcto. Como ejercicio, sugerimos al lector que analice los resultados en los casos de que la premisa 2 fuese:

- a) "si Bernardo no se hace monje, Florinda se suicida", y
- b) "si Florinda se suicida, Bernardo no se hace monje".

Ejemplo 1.4.4. Veamos ahora un razonamiento con tres premisas:

$P1$: No hay judíos en la cocina

$P2$: Ningún gentil dice "sphoonj"

$P3$: Todos mis sirvientes están en la cocina

C : Mis sirvientes no dicen nunca "sphoonj"

Aunque el modo más natural de formalizar este razonamiento sería en lógica de predicados, y así lo veremos en el capítulo 4, también podemos hacerlo en lógica de proposiciones. En efecto, definiendo las variables proposicionales:

j : Alguien es judío

c : Alguien está en la cocina

d : Alguien dice "sphoonj"

s : Alguien es sirviente

podemos formalizar así el razonamiento:

$P1: j \rightarrow \neg c$ ("si alguien es judío entonces no está en la cocina")

$P2: \neg j \rightarrow \neg d$ ("si alguien no es judío entonces no dice 'sphoonj'")

$P3: s \rightarrow c$ ("si alguien es sirviente entonces está en la cocina")

$C: s \rightarrow \neg d$ ("si alguien es sirviente entonces no dice 'sphoonj'")

Enlazando las tres premisas entre sí mediante conjunciones y a la conclusión mediante un condicional, obtenemos la siguiente sentencia, expresión formal del razonamiento:

$$[(j \rightarrow \neg c) \wedge (\neg j \rightarrow \neg d) \wedge (s \rightarrow c)] \rightarrow (s \rightarrow \neg d)$$

Calculemos las distintas evaluaciones de la sentencia. En la siguiente tabla de verdad se indican los pasos intermedios y el resultado.

j	c	d	s	$P1$	$P2$	$P3$	$P1 \wedge P2 \wedge P3$	C	S
0	0	0	0	1	1	1	1	1	1
0	0	0	1	1	1	0	0	1	1
0	0	1	0	1	0	1	0	1	1
0	0	1	1	1	0	0	0	0	1
0	1	0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1
0	1	1	0	1	0	1	0	1	1
0	1	1	1	1	0	1	0	0	1
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	1	1	1	1	1	1
1	0	1	1	1	1	0	0	0	1
1	1	0	0	0	1	1	0	1	1
1	1	0	1	0	1	1	0	1	1
1	1	1	0	0	1	1	0	1	1
1	1	1	1	0	1	1	0	0	1

Vemos que la sentencia es una tautología, y, por consiguiente, el razonamiento es válido. No obstante, en casos como éste la lógica de proposiciones empieza a ser un lenguaje demasiado limitado para representar todos los matices del lenguaje natural. Concretamente, aquí hemos hecho una pequeña "trampa": tal como se han definido las variables proposicionales, lo representado por " $j \rightarrow \neg c$ " sería: "si alguien es judío entonces no es cierto que alguien esté en la cocina", lo cual no es exactamente lo que queremos decir, y lo mismo con las otras tres sentencias.

Lo que ocurre es que con la palabra "alguien" del lenguaje natural se representa a un miembro cualquiera de un colectivo, es decir, formalmente, a una *variable* (no una variable proposicional). Nuestras proposiciones hacen referencia a propiedades de tales variables que determinan subconjuntos del colectivo, y esto es algo que no se puede formalizar en lógica de proposiciones. Si aquí hemos podido llegar a una formalización ha sido "fijando" ese "alguien", es decir, considerando que en todas las premisas y en la conclusión se trata del mismo individuo (como si, en vez de "alguien", hubiésemos dicho, por ejemplo, "Isaac") y suponiendo implícitamente que si el razonamiento es válido para un individuo cualquiera, lo es para todos.

Ejemplo 1.4.5. Supongamos que un político declara en una rueda de prensa:

P1: 'Si los impuestos suben, la inflación bajará, pero solamente si la peseta no se devalúa'.

Al día siguiente afirma en el Congreso:

P2: 'Si la inflación baja, o si la peseta no se devalúa, los impuestos no subirán'.

Y esa misma noche manifiesta en una entrevista por televisión:

P3: 'O bien baja la inflación y se devalúa la peseta, o bien los impuestos deben subir'.

Nuestro político elabora luego un informe para el Ministerio de Economía que concluye con la siguiente recomendación:

C: 'Los impuestos deben subir, pero la inflación bajará y la peseta no se devaluará'.

Preguntémonos (imaginándonos por un momento en el papel del ministro) si tal conclusión es consistente con lo que este señor había dicho anteriormente. En principio, no cuestionamos la verdad o la falsedad de las premisas planteadas: si alguna es falsa, la conclusión no tiene por qué ser verdadera, pero si las tres son verdaderas, la conclusión también debe serlo. Es decir, para que la inferencia de

la conclusión sea correcta, la sentencia $(P1 \wedge P2 \wedge P3) \rightarrow C$ deberá ser una tautología.

Así pues, definiendo las variables proposicionales p , q y r con esta interpretación:

$i(p)$ ='los impuestos suben'
 $i(q)$ ='la inflación baja'
 $i(r)$ ='la peseta no se devalúa'

la sentencia a analizar será:

$$[(p \rightarrow (q \leftrightarrow r)) \wedge ((q \vee r) \rightarrow \neg p) \wedge ((q \wedge \neg r) \vee p)] \rightarrow (p \wedge q \wedge r)$$

Dejamos al cuidado del lector la construcción de la tabla de verdad correspondiente. Podrá comprobar que no se trata de una tautología (por lo que la conclusión no es correcta), y descubrir los dos casos que invalidan el razonamiento.

1.5. Cálculo, sistema axiomático y sistema inferencial

Veremos en el siguiente apartado que la lógica de proposiciones es un lenguaje formal (en el sentido definido en el capítulo 1), complementado con un conjunto de axiomas y de reglas de transformación.

Un *cálculo* es la estructura formal de un lenguaje, abstrayendo el significado; se convierte en un lenguaje cuando se evalúan sus símbolos y sus construcciones (es decir, se les atribuye un significado, se les pone en relación con los objetos que designan).

El cálculo es, además, un *sistema axiomático* cuando se construye sobre la base de unos *axiomas*, o construcciones que se admiten como verdaderas en el lenguaje o lenguajes de los que procede el cálculo (o, lo que es lo mismo, en todos los lenguajes que se pueden formalizar con él).

Para definir un sistema axiomático hay que especificar:

- el *alfabeto*, o conjunto de símbolos válidos en el cálculo;
- las *reglas de formación*, que permiten derivar sentencias (cadenas de símbolos válidas o correctas);
- los *axiomas*;
- las *reglas de transformación*, que permiten obtener o *demonstrar* determinadas sentencias a partir de los axiomas (*sentencias demostrables*, *teoremas* o *leyes*).

Los axiomas y los teoremas constituyen las *tesis* del sistema axiomático, y pertenecen al lenguaje del cálculo; las reglas de formación y de transformación pertenecen al metalenguaje del cálculo.

La formulación de los axiomas no es arbitraria ni fácil. Tiene que satisfacer dos exigencias metalógicas imprescindibles:

- a) el cálculo resultante debe ser *completo*, es decir, todas las sentencias *verdaderas* (o sea, todas las tautologías) relativas a las teorías (o lenguajes) que el cálculo pretende formalizar tienen que poder demostrarse a partir de los axiomas.
- b) el cálculo debe ser *consistente*, es decir, que no se puedan demostrar con él sentencias no verdaderas.

Conviene, además, que los axiomas sean independientes, es decir, que ninguno de ellos pueda demostrarse a partir de los restantes. Esta condición no es fácil de comprobar. Prueba de ello es que en el sistema PM, que veremos en el apartado 2.5, se proponían inicialmente cinco axiomas, y sólo años más tarde se comprobó que uno era redundante (se podía demostrar a partir de los otros cuatro).

El lector más interesado por las aplicaciones informáticas de la lógica que por ésta en sí misma puede pensar que todo esto es de poca utilidad. Pero no es así. En las aplicaciones, efectivamente, no tiene un interés especial la demostración de teoremas a partir de unos axiomas. Lo realmente interesante es mecanizar los procesos de inferencia. Es decir, dadas unas premisas o hechos comprobados como verdaderos *en una determinada situación*, poder deducir conclusiones para esa situación. Esto es lo que llamaremos un *sistema inferencial*: un conjunto de reglas (pertenecientes al metalenguaje de la lógica) que permiten ejecutar inferencias y unas metarreglas (pertenecientes al metametalenguaje) que dicen cómo aplicarlas. Ahora bien, veremos en el apartado 5 que los teoremas del sistema axiomático son precisamente la base de las reglas que permiten realizar inferencias. Por otra parte, para acercarnos al ideal de "poder asegurar" el correcto funcionamiento de los programas es preciso trabajar sobre una base teóricamente sólida y rigurosa. (La pragmática de la informática dice que nunca se puede asegurar que un programa funciona correctamente,⁴ pero poco a poco se avanza hacia la posibilidad de la demostración formal de la corrección de los programas. Véanse a este respecto el apartado 6 del capítulo 3 del tema "Algoritmos" y el apartado 4.1 del capítulo 5 del tema "Lenguajes").

⁴ Hay una antigua "ley" de la informática práctica que dice: "*Todos los programas reales contienen errores mientras no se demuestre lo contrario, lo cual es imposible*" (Gilb, 1975).

2. Sintaxis

2.1. Alfabeto

Entramos ya a definir formalmente la lógica de proposiciones como un lenguaje, y en este apartado presentamos el cálculo de proposiciones. El alfabeto que utilizaremos será el formado por los siguientes símbolos:

- Variables proposicionales: p, q, r, s, t, \dots (En caso necesario, con subíndices numéricos, y, en los ejemplos, cualquiera otra minúscula, si conviene por motivos nemotécnicos).
- Conectivas: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ (Estas serán las más utilizadas, aunque pueden definirse otras, como veremos).
- Símbolos de puntuación: $(,), [,], \{, \}$ (Lo mismo que en el lenguaje más familiar del álgebra, los utilizaremos para evitar ambigüedades).
- Metasímbolos: Como indica su nombre, no son, propiamente, símbolos del lenguaje. Los utilizaremos para abreviar determinadas expresiones, y serán los siguientes:

A, B, C, \dots, S, \dots para representar cualquier sentencia del lenguaje (también, en caso necesario, con subíndices).

k para representar cualquier conectiva binaria.

l para representar una variable proposicional sola o con el símbolo de negación delante. (Esto es lo que se llama un *literal*: literal *positivo* si corresponde a una variable proposicional sin negación, y *negativo* en caso contrario).

2.2. Expresiones y sentencias

Definición 2.2.1. Llamaremos *expresión* o *cadena* a toda secuencia finita de símbolos del alfabeto. Por ejemplo:

$$(p \vee \neg q) \rightarrow r$$

$$rv) \rightarrow pq \neg ($$

$$(p \vee q \neg) \rightarrow r$$

son tres expresiones. Intuitivamente, se aprecia que sólo la primera puede "significar" algo; diremos que, de las tres, sólo ella es una sentencia. Definamos formalmente este concepto:

Definición 2.2.2. Llamaremos *secuencia de formación* a toda secuencia finita de expresiones, A_1, A_2, \dots, A_n , en la que cada A_i satisface al menos una de las tres condiciones siguientes:

- a) A_i es una variable proposicional;
- b) existe un j menor que i tal que A_i es el resultado de anteponer el símbolo " \neg " a A_j ;
- c) existen j y h , ambos menores que i , tales que A_i es el resultado de enlazar A_j y A_h con k (cualquier conectiva binaria).

Definición 2.2.3. Llamaremos *sentencia* a toda expresión A_n tal que existe una secuencia de formación A_1, A_2, \dots, A_n . Y ésta será una *secuencia de formación de A_n* (puede no ser la única). Obsérvese que, según esta definición, todas las expresiones que componen una secuencia de formación son sentencias.

Una forma equivalente de definir el concepto de sentencia es enunciando tres *reglas de formación* (correspondientes a las tres condiciones anteriores):

RF1: Una variable proposicional es una sentencia.

RF2: Si A es una sentencia, $\neg A$ también lo es.

RF3: Si A y B son sentencias, AkB también lo es.

Estas tres reglas constituyen una *definición recursiva* de sentencia. (Recursiva, porque en ella se alude al concepto definido). Son reglas gramaticales que definen la sintaxis de la lógica de proposiciones. En el tema "Lenguajes", capítulo 5, apartado 3, veremos que estas reglas, aquí enunciadas utilizando el castellano como metalenguaje, pueden formalizarse con un metalenguaje llamado "notación BNF", muy utilizado en informática para la definición rigurosa de lenguajes de programación.

En algunos textos el lector puede encontrar que a las sentencias se les llama "wff". Son siglas de "well formed formula" (fórmulas bien formadas).

2.3. Sentencias equivalentes

El concepto de equivalencia entre sentencias tiene que ver con la semántica, y lo definiremos en el apartado 3.5. De momento, digamos que dos sentencias son equivalentes si, dada una interpretación cualquiera de las variables proposicionales que las componen, los significados (interpretaciones) de ambas sentencias son idénticos. Aunque esta definición no sea muy precisa, haremos ya uso de algunas equivalencias sencillas, que nos permitirán sustituir unas sentencias por otras. Por ejemplo, cuatro equivalencias triviales (intuitivamente) son:

- a) $A \vee B$ es equivalente a $B \vee A$
- b) $A \wedge B$ es equivalente a $B \wedge A$
- c) $A \vee (B \vee C)$ es equivalente a $A \vee (B \vee C)$ (y se puede escribir $A \vee B \vee C$)
- d) $A \wedge (B \wedge C)$ es equivalente a $A \wedge (B \wedge C)$ (y se puede escribir $A \wedge B \wedge C$)

A cada una de estas equivalencias le corresponde un teorema (concepto que definiremos en el apartado 2.5): las sentencias

- a) $(p \vee q) \leftrightarrow (q \vee p)$
- b) $(p \wedge q) \leftrightarrow (q \wedge p)$
- c) $(p \vee (q \vee r)) \leftrightarrow (p \vee (q \vee r))$
- d) $(p \wedge (q \wedge r)) \leftrightarrow (p \wedge (q \wedge r))$

pueden demostrarse como teoremas, siguiendo el procedimiento que veremos en la sección 2.5.3.

2.4. Conectivas primitivas y definidas

De las cuatro conectivas binarias definidas en el alfabeto, basta con una. Si, por ejemplo, nos quedamos con las conectivas primitivas " \neg " y " \vee ", las tres restantes se definen del siguiente modo:

- a) Definición de " \wedge ": Una sentencia de la forma

$$A \wedge B$$

es equivalente a una sentencia de la forma

$$\neg(\neg A \vee \neg B)$$

- b) Definición de " \rightarrow ": Una sentencia de la forma

$$A \rightarrow B$$

es equivalente a una sentencia de la forma

$$\neg A \vee B$$

- c) Definición de " \leftrightarrow ": Una sentencia de la forma

$$A \leftrightarrow B$$

es equivalente a una sentencia de la forma

$$\neg(\neg(\neg A \vee B) \vee \neg(\neg B \vee A))$$

o bien, si se permite utilizar la conectiva previamente definida " \wedge ",

$$(\neg A \vee B) \wedge (\neg B \vee A)$$

A diferencia de las escritas en el apartado anterior, estas equivalencias *no* corresponden a teoremas: *son definiciones* de las conectivas " \wedge ", " \rightarrow " y " \leftrightarrow " en función de las conectivas primitivas " \neg " y " \vee ".

2.5. Axiomas, demostraciones y teoremas

2.5.1. Axiomas

El sistema axiomático más conocido es el llamado "PM", siglas derivadas del título de una obra clásica: "*Principia Mathematica*", de Whitehead y Russell. Utiliza cuatro axiomas:

$$A1. (p \vee p) \rightarrow p$$

$$A2. q \rightarrow (p \vee q)$$

$$A3. (p \vee q) \rightarrow (q \vee p)$$

$$A4. (p \rightarrow q) \rightarrow [(r \vee p) \rightarrow (r \vee q)]$$

Como hemos avanzado en el apartado 1.5, la idea es que con estos axiomas y unas reglas de transformación, junto con las definiciones dadas más arriba que permiten sustituir unas conectivas por otras, se demuestran teoremas. Vamos a definir las reglas de transformación y el proceso de demostración, pero antes hemos de introducir una operación previa: la sustitución.

2.5.2. Sustitución

Definición 2.5.2.1. Dadas unas variables proposicionales, p_1, p_2, \dots, p_n y unas sentencias cualesquiera, B_1, B_2, \dots, B_n , llamaremos *sustitución* a un conjunto de pares ordenados

$$s = \{B_1/p_1, B_2/p_2, \dots, B_n/p_n\}$$

Definición 2.5.2.2. Dadas una sentencia A que contiene (quizás, entre otras) las variables proposicionales p_1, p_2, \dots, p_n y una sustitución s , la *operación de sustitución* en A de esas variables proposicionales por las sentencias B_1, B_2, \dots, B_n , consiste en poner en A , en todos los lugares donde aparezca la variable p_i , la sentencia B_i que le corresponde de acuerdo con s , y esto para todo $i=1\dots n$. El resultado es una expresión que representaremos como As .

Teorema 2.5.2.3. Si A es una sentencia y s una sustitución, As es una sentencia. La demostración, que dejamos al lector, es sencilla: basta con ver que si tanto A como todas las B_i tienen secuencias de formación, entonces As tiene también, al menos, una secuencia de formación. Es interesante observar que este teorema, como otros que iremos dando, es, en realidad, un "metateorema" para la lógica de proposiciones.

2.5.3. Demostraciones y teoremas

Definición 2.5.3.1. Llamaremos *demostración* (o *prueba formal*) a toda secuencia finita de sentencias A_1, A_2, \dots, A_n , en la que cada A_i satisface, al menos, una de las cuatro condiciones siguientes:

- a) A_i es un axioma;
- b) Existe algún j menor que i y alguna sustitución s tal que A_i es el resultado de la sustitución s en A_j , es decir, A_i es lo mismo que $A_j s$;
- c) Existen h y j menores que i tales que A_i es lo mismo que $A_h \wedge A_j$;
- d) Existen h y j menores que i tales que A_h es lo mismo que $A_j \rightarrow A_i$.

Definición 2.5.3.2. Llamaremos *teorema* a toda sentencia A_n que no es un axioma y que es tal que existe una demostración A_1, A_2, \dots, A_n , y diremos que ésta es una *demostración de A_n* (puede no ser única). Obsérvese que todas las sentencias que componen una demostración son o bien axiomas o bien teoremas.

Definición 2.5.3.3. Llamaremos *tesis* (o *ley*) del sistema axiomático a cualquier sentencia que sea o bien un axioma o bien un teorema. Utilizaremos la notación " $\vdash A$ " para indicar que " A es una tesis".

De modo similar a lo que hacíamos al definir las sentencias, podemos dar una definición recursiva de tesis mediante cuatro reglas de transformación (correspondientes a las cuatro condiciones anteriores):

2.5.4. Reglas de transformación

RT1. Si A es un axioma, entonces A es una tesis

RT2. (Regla de sustitución). Si A es una tesis en la que aparecen p_1, p_2, \dots, p_n y B_1, B_2, \dots, B_n son sentencias, entonces $A \{B_1/p_1, \dots, B_n/p_n\}$ es una tesis.

RT3. (Regla de unión). Si A y B son tesis entonces $A \wedge B$ es una tesis.

RT4. (Regla de separación). Si A y $A \rightarrow B$ son tesis entonces B es una tesis.

2.5.5. Ejemplos de demostraciones en el sistema PM

Teorema 1: $(p \rightarrow q) \rightarrow [(r \rightarrow p) \rightarrow (r \rightarrow q)]$

Demostración:

1. $(p \rightarrow q) \rightarrow [(\neg r \vee p) \rightarrow (\neg r \vee q)]$ (Por sustitución A4 $\{\neg r/r\}$)
2. $(p \rightarrow q) \rightarrow [(r \rightarrow p) \rightarrow (r \rightarrow q)]$ (Por definición de la conectiva " \rightarrow ")

Teorema 2: $p \rightarrow (p \vee p)$

Demostración: Simplemente, por sustitución A2 $\{p/q\}$.

Teorema 3: $p \rightarrow p$

Demostración:

1. $[(p \vee p) \rightarrow p] \rightarrow [(p \rightarrow (p \vee p)) \rightarrow (p \rightarrow p)]$ (Por sustitución en el Teorema 1: T1 $\{(p \vee p)/p, p/q, p/r\}$)
2. $[p \rightarrow (p \vee p)] \rightarrow (p \rightarrow p)$ (Por separación de A1 en la sentencia anterior)
3. $p \rightarrow p$ (Por separación del Teorema 2 en la sentencia anterior)

Otra posible demostración:

1. $(p \vee p) \rightarrow (p \vee p)$ (Por sustitución A3 $\{p/q\}$)
2. $p \rightarrow p$ (Por sustitución $\{p/p \vee p\}$ en la anterior)

Teorema 4: $\neg p \vee p$ (Principio del tercio excluso)

Demostración: Por definición de la conectiva " \rightarrow " en el Teorema 3.

Para no recargar la notación, no hemos escrito el símbolo " \vdash ", lo que podríamos haber hecho delante de cada una de las sentencias que han ido apareciendo en las demostraciones.

2.5.6. Algunos teoremas útiles

Entre la infinidad de teoremas que pueden demostrarse están, por ejemplo:

- La *ley de modus ponendo ponens* (o *modus ponens*):

$$[p \wedge (p \rightarrow q)] \rightarrow q$$

Corresponde al tipo de razonamiento según el cual, dado un condicional y afirmando ("ponendo") el antecedente, se puede afirmar ("ponens") el consecuente, pero esto ya es una "regla de inferencia", de la que hablaremos más adelante.

- La *ley de modus tollendo tollens* (o *modus tollens*):

$$[\neg q \wedge (p \rightarrow q)] \rightarrow \neg p$$

Corresponde al tipo de razonamiento según el cual, dado un condicional y negando ("tollendo") el consecuente, se puede negar ("tollens") el antecedente, otra regla de inferencia.

- Las *leyes de transitividad*:

$$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$$

$$[(p \leftrightarrow q) \wedge (q \leftrightarrow r)] \rightarrow (p \leftrightarrow r)$$

Corresponden a los "silogismos hipotéticos" de la lógica clásica.

- Las *leyes de inferencia de la alternativa*, o de los *silogismos disyuntivos*:

$$[\neg p \wedge (p \vee q)] \rightarrow q$$

$$[p \wedge (\neg p \vee \neg q)] \rightarrow \neg q$$

- La *ley del dilema constructivo*:

$$[(p \rightarrow q) \wedge (r \rightarrow s) \wedge (p \vee r)] \rightarrow (q \vee s)$$

- Las *leyes de Morgan*:

$$\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$$

$$\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$$

Teoremas como estos dos últimos, de la forma $A \leftrightarrow B$, establecen una equivalencia entre las sentencias A y B , como veremos en el apartado 3.5. Otros de la misma forma, que utilizaremos más adelante, son:

- La *ley de doble negación*:

$$\neg\neg p \leftrightarrow p$$

- La *ley de reducción al absurdo*:

$$[\neg p \rightarrow (q \wedge \neg q)] \leftrightarrow p$$

- Las leyes de distributividad:

$$[(p \wedge q) \vee r] \leftrightarrow [(p \vee r) \wedge (q \vee r)]$$

$$[(p \vee q) \wedge r] \leftrightarrow [(p \wedge r) \vee (q \wedge r)]$$

$$[(p \rightarrow q) \vee r] \leftrightarrow [(p \vee r) \rightarrow (q \vee r)]$$

$$[p \rightarrow (q \vee r)] \leftrightarrow [(p \rightarrow q) \vee (p \rightarrow r)]$$

$$[p \rightarrow (q \wedge r)] \leftrightarrow [(p \rightarrow q) \wedge (p \rightarrow r)]$$

$$[p \rightarrow (q \rightarrow r)] \leftrightarrow [(p \rightarrow q) \rightarrow (p \rightarrow r)]$$

3. Semántica

Las sentencias del lenguaje lógico permiten representar conocimientos acerca de la realidad. Como paso previo a la formalización de esos conocimientos se establece un *modelo conceptual de la realidad*, o *conceptualización*, que acota los objetos y aspectos de la realidad que las sentencias van a representar. En el caso de la lógica de proposiciones, una conceptualización es, simplemente, un conjunto de proposiciones.

La semántica establece, como sabemos (capítulo 1, apartado 5.4), una correspondencia entre las construcciones del lenguaje y los objetos que designan. En este caso, entre las sentencias del cálculo de proposiciones y los elementos de la conceptualización.

Representaremos por CP al conjunto (finito) de elementos de la conceptualización, por P a un conjunto (finito) de variables proposicionales, por S al conjunto (infinito) de sentencias que pueden construirse con P , por S_n al subconjunto (infinito) de S cuyas sentencias tienen exactamente n variables proposicionales, y por S^n al subconjunto (finito si n es finito) de S cuyas sentencias tienen un número de conectivas igual o inferior a n .

3.1. Interpretaciones y evaluaciones

Definición 3.1.1. Una *interpretación de variables proposicionales*, i , es una función que hace corresponder a cada elemento de P un elemento de CP .

Toda interpretación de variables proposicionales lleva asociada una evaluación de las mismas. Vamos a definir este concepto de manera muy general (aplicable a lógicas no binarias).

Definición 3.1.2. Un *conjunto de valores de verdad* es un conjunto de símbolos, V , de cardinalidad igual o superior a 2, con una relación de orden definida entre sus elementos y unos extremos superior e inferior, y cerrado bajo

determinadas operaciones definidas en él. A cada conectiva definida en el alfabeto debe corresponder una operación en V .

Obsérvese que, en principio, no imponemos más restricción a V que la de tener dos o más elementos (puede, incluso, tener infinitos), y de tener definidas en él (sin fijar cómo) una operación por cada conectiva, una relación de orden y unos extremos superior (que corresponde a la "certeza absoluta") e inferior (la "falsedad absoluta").

La evaluación de una variable proposicional (o, en general, de una sentencia) consiste en atribuirle un valor de verdad. Ahora bien, una misma sentencia puede evaluarse con un valor u otro, dependiendo de la interpretación. La evaluación es, pues, un concepto *relativo* (dependiente de la interpretación).

Definición 3.1.3. Dados un conjunto de variables proposicionales, P , un conjunto de valores de verdad, V , y una interpretación, i , una *evaluación* de P relativa a i , e_i , es una función total de P en V : $(\forall p \in P) (e_i(p) \in V)$.

De este modo, cuando damos una evaluación (para una determinada interpretación), lo que hacemos es asignar a cada variable proposicional un elemento de V . Para evaluar sentencias, tendremos que establecer una extensión de i para que su dominio sea S en lugar de P .

Teorema 3.1.4. Dada una evaluación e_i , existe una (y sólo una) extensión del dominio de e_i a S , a la que llamaremos E_i , tal que satisface estas dos condiciones:

- a) $(\forall A \in S) (E_i(\neg A) = \neg E_i(A))$
- b) $(\forall A, B \in S) (E_i(A \mathbin{k} B) = E_i(A) \mathbin{k} E_i(B))$

(Adviértase que, aunque los símbolos utilizados sean los mismos, " \neg " y " k " representan distintas cosas en el primero y en el segundo miembro de esas igualdades: en el primero, son las conectivas utilizadas en el alfabeto, que se aplican a variables o a sentencias para obtener otras sentencias; en el segundo, son las operaciones definidas en el conjunto V).

Demostración:

- a) Existencia. La demostración de que existe al menos una E_i que cumple esas condiciones puede hacerse por inducción sobre n (número de conectivas de las sentencias) y teniendo en cuenta que toda sentencia tiene que estar construida según las reglas de formación dadas en el apartado 2.2. Si $A \in S^n$, escribiremos $E_i^n(A)$ en lugar de $E_i(A)$ para indicar explícitamente que se trata de una evaluación de una sentencia con n conectivas:

- Para $n=0$, $E_i^0(A) = e_i(A)$;
- Supuesto que existe E_i^n , definimos E_i^{n+1} así:
 si $A \in S^n$, $E_i^{n+1}(\neg A) = \neg E_i^n(A)$;
 si $A, B \in S^n$, $E_i^{n+1}(A \wedge B) = E_i^n(A) \wedge E_i^n(B)$

De este modo, hemos construido una aplicación E_i^{n+1} que cumple las condiciones impuestas, y que está definida para toda sentencia de S^{n+1} .

- b) Unicidad. E_i^0 es única, puesto que es igual a e_i . Y si E_i^n es única, E_i^{n+1} también lo será, puesto que se ha construido de manera única.

Corolario: E_i está unívocamente determinada por e_i . Esto quiere decir que, dada una evaluación de variables proposicionales, puede calcularse la evaluación de cualquier sentencia construida con esas variables proposicionales.

Este teorema justifica matemáticamente la construcción de tablas de verdad que hacíamos en los ejemplos del apartado 1.4. Pero obsérvese que ahora tenemos un marco mucho más general, puesto que el teorema se aplica independientemente del número de elementos que tenga V .

3.2. Satisfacción

En el apartado anterior hemos definido el concepto de evaluación de una manera general, lo cual permite aplicarlo a lógicas en las que las proposiciones (y las sentencias) pueden tomar valores intermedios entre "verdadero" y "falso". Veremos algunos de estos tipos de lógica en el capítulo 5. De momento, y hasta entonces, nos limitaremos a la lógica clásica, o "binaria".

Definición 3.2.1. Una *evaluación binaria* es una evaluación E_i sobre un conjunto de valores de verdad V que sólo tiene dos elementos, que llamaremos "0" (o "falso") y "1" (o "verdadero") y (además de otras que luego veremos), cinco operaciones definidas en él de la siguiente forma:

$$\neg(0) = 1; \neg(1) = 0;$$

$$0 \wedge 0 = 0; 0 \wedge 1 = 0; 1 \wedge 0 = 0; 1 \wedge 1 = 1;$$

$$0 \vee 0 = 0; 0 \vee 1 = 1; 1 \vee 0 = 1; 1 \vee 1 = 1;$$

$$0 \rightarrow 0 = 1; 0 \rightarrow 1 = 1; 1 \rightarrow 0 = 0; 1 \rightarrow 1 = 1;$$

$$0 \leftrightarrow 0 = 1; 0 \leftrightarrow 1 = 0; 1 \leftrightarrow 0 = 0; 1 \leftrightarrow 1 = 1.$$

El extremo inferior de V es "0", el superior es "1", y la relación de orden es " $0 < 1$ ".

Esta definición de las operaciones es la misma que ya habíamos visto y comentado en el apartado 1.3. Lo que hemos hecho ahora ha sido particularizar, partiendo de una definición formal y más general. Por ejemplo, si $P=\{p,q\}$, el número de interpretaciones posibles es infinito, pero sólo hay cuatro evaluaciones binarias diferentes, las que dimos en el apartado 1.3. Pero si V fuese un conjunto con más de dos elementos, entonces tendríamos más evaluaciones, como veremos en el capítulo 5, apartado 3.

En el caso binario, si consideramos, por ejemplo, la sentencia $p \vee q$, la extensión de $(e_1(p) = 0; e_1(q) = 1)$ aplicada a esta sentencia, de acuerdo con la construcción definida en la demostración del teorema 3.1.4, será:

$$E_1(p \vee q) = E_1(p) \vee E_1(q) = e_1(p) \vee e_1(q) = 0 \vee 1 = 1$$

e igualmente podrían calcularse todas las evaluaciones dadas en la tabla del apartado 1.3.

Si $\text{card}(P)=n$, entonces habrá 2^n evaluaciones binarias de las variables proposicionales, $e_j: P \rightarrow \{0, 1\}$ ($j= 0 \dots 2^n-1$), y, para cada una de ellas, cualquier sentencia A tendrá una evaluación, $E_j(A)$.

Definición 3.2.2. Una interpretación de variables proposicionales, i , *satisface* a una sentencia A si (y sólo si) $E_i(A) = 1$. También diremos que A *se satisface* con la interpretación i .

Volviendo al primer ejemplo del apartado 1.3., consideremos las sentencias:

$$A1: p \vee q$$

$$A2: p \wedge q$$

y las interpretaciones y evaluaciones:

$$i_1(p) = \text{la nieve es blanca}; e_1(p) = 1$$

$$i_1(q) = \text{el carbón es negro}; e_1(q) = 1$$

$$i_2(p) = \text{la nieve es blanca}; e_2(p) = 1$$

$$i_2(q) = \text{el carbón es blanco}; e_2(q) = 0$$

El lector puede comprobar que $A1$ se satisface con ambas interpretaciones, mientras que $A2$ sólo se satisface con i_1 .

3.3. Tautologías y contradicciones

Por tanto, la satisfacción de una sentencia depende, en general, de la interpretación dada a sus variables proposicionales. Pero hay algunas sentencias que siempre se satisfacen, sea cual sea la interpretación; y otras que no se satisfacen nunca. A las primeras se les llama "tautologías", y a las segundas, "contradicciones". Más formalmente:

Definición 3.3.1. Diremos que una sentencia A es una *tautología* si (y sólo si) $(\forall i) (E_i(A) = 1)$. Utilizaremos la notación " $\models A$ " para indicar que " A es una tautología".

Definición 3.3.2. Diremos que una sentencia A es una *contradicción* si (y sólo si) $(\forall i) (E_i(A) = 0)$.

En la lógica binaria, que es la que estamos considerando, y por definición de la operación " \neg " en el conjunto $V = \{0, 1\}$, si $\models A$ entonces $\neg A$ es una contradicción, y viceversa: si A es una contradicción entonces $\models (\neg A)$. Como puede comprobarse fácilmente, $\neg p \vee p$ es una tautología, y $\neg(\neg p \vee p)$ es una contradicción. Del mismo modo, $\neg p \wedge p$ es una contradicción, y $\neg(\neg p \wedge p)$ una tautología.

3.4. Completitud y consistencia de un sistema axiomático

Ahora podemos expresar con mayor rigor algo que introducíamos en el apartado 1.5:

Definición 3.4.1. Diremos que un sistema axiomático es *completo* si toda sentencia A que sea una tautología es también una tesis. Es decir, para toda A , si $\models A$ entonces $\vdash A$.

Definición 3.4.2. Diremos que un sistema axiomático es *consistente* si toda sentencia A que sea una tesis es también una tautología. Es decir, para toda A , si $\vdash A$ entonces $\models A$.

Puede demostrarse que el sistema PM es completo y consistente. El lector puede comprobar, por ejemplo (construyendo las tablas de verdad), que tanto los axiomas como los teoremas que hemos demostrado o citado son tautologías.

3.5. Equivalencia entre sentencias

Definición 3.5.1. Dadas dos sentencias A y B construidas con las mismas variables proposicionales, P , diremos que A es *equivalente* a B , y escribiremos $A \equiv B$ si (y sólo si)

$$(\forall i) (E_i(A) = E_i(B))$$

Es inmediato comprobar que la relación es, en efecto, una relación de equivalencia, es decir, que es reflexiva, simétrica y transitiva.

Como ejercicio, el lector también puede comprobar que las equivalencias indicadas en los apartados de números 2.3 y 2.4 cumplen los requisitos de esta definición.

Como toda relación de equivalencia, " \equiv " particiona al conjunto (infinito) de sentencias construidas con n variables en un conjunto de clases de equivalencia, $Q_n = S_n / \equiv$. Si el número de evaluaciones diferentes de las variables proposicionales es finito (lo cual ocurre si V es finito) entonces Q_n es también finito. Por ejemplo, con $n=2$ (dos variables proposicionales, p y q) y con $V=\{0,1\}$ resultan dieciséis clases de equivalencia, $Q_2 = \{C_0, C_1, \dots, C_{15}\}$ cuyas evaluaciones son las siguientes:

p	q	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Así, todas las sentencias cuyas cuatro evaluaciones sean las de la clase $C_1(p \wedge q, \neg(\neg p \vee \neg q), \neg(p \rightarrow \neg q), \text{etc.})$ son equivalentes entre sí.

Teorema 3.5.2. En un sistema axiomático consistente, $A \equiv B$ si y sólo si $\vdash(A \leftrightarrow B)$. Es decir, las sentencias A y B son equivalentes si y sólo si la sentencia $A \leftrightarrow B$ es una tesis.

Demostración:

- Si $\vdash(A \leftrightarrow B)$ y el sistema axiomático es consistente, entonces $\models(A \leftrightarrow B)$, es decir, $A \leftrightarrow B$ es una tautología. Por definición del bicondicional, $A \leftrightarrow B$ es verdadera (y sólo es verdadera) si A y B son ambas verdaderas o ambas falsas. Por tanto, si $A \leftrightarrow B$ es una tautología (verdadera para todas las evaluaciones) entonces A y B han de tener las mismas evaluaciones para todas y cada una de las evaluaciones de las variables proposicionales. Y ésta es precisamente la definición de equivalencia entre A y B . Concluimos así que si $\vdash(A \leftrightarrow B)$ entonces $A \equiv B$.
- Si $A \equiv B$, por la misma definición 3.5.1 y por la definición del bicondicional, $\models(A \leftrightarrow B)$. Y si el sistema axiomático es completo ello implica que $\vdash(A \leftrightarrow B)$. Concluimos así que si $A \equiv B$ entonces $\vdash(A \leftrightarrow B)$.

3.6. Las conectivas binarias

Hasta ahora venimos utilizando cuatro conectivas diádicas o binarias (binarias, en el sentido de que enlazan dos variables proposicionales; además, e independientemente de ello, estamos considerando una lógica binaria, en el sentido de que $V=\{0,1\}$). Hemos visto que, gracias a algunas equivalencias (apartado 2.4), podemos limitarnos a usar una sola de ellas, junto con la negación. Pero también podemos hacer lo contrario: definir nuevas conectivas. Veamos cuántas. Si comparamos la tabla del apartado 1.3, en la que se definía el significado de las conectivas utilizadas, con la tabla del apartado anterior, en la que se muestran las evaluaciones de las dieciséis clases de equivalencia de las sentencias construidas con dos variables proposicionales, vemos inmediatamente que las conectivas " \wedge ", " \vee ", " \rightarrow " y " \leftrightarrow " se corresponden con las clases de equivalencia C_1 , C_7 , C_{13} y C_9 , respectivamente. Y para cada una de las otras doce clases podemos definir una nueva conectiva binaria. Por ejemplo, C_6 correspondería a la disyunción exclusiva, para la que a veces se utiliza el símbolo " \oplus ".

Las equivalencias dadas en el apartado 2.4 nos permitían representar las clases de equivalencia C_1 , C_{13} y C_9 mediante sentencias que sólo utilizan las conectivas " \neg " y " \vee ". Lo mismo puede hacerse para las otras clases. Por ejemplo, la clase C_0 se representaría mediante $\neg(p \vee \neg p)$, la C_2 mediante $\neg(\neg p \vee q)$, etc. Igualmente, podríamos trabajar con la pareja " \neg ", " \wedge ", o con la " \neg ", " \leftrightarrow ", etc. Pero también es posible representar cualquier clase de equivalencia utilizando solamente una conectiva. Ello puede hacerse con dos de las dieciséis conectivas binarias:

- a) La "incompatibilidad", o "negación alternativa", o "NAND", que corresponde a la clase C_{14} , y que representaremos con el símbolo " $|$ ".
- b) La "negación conjunta", o "NOR", que corresponde a la clase C_8 , y que representaremos con el símbolo " \downarrow ".

Por ejemplo, las equivalencias que permiten sustituir a las conectivas más utilizadas por una cualquiera de éstas son:

$$\text{Para } C_{12}: \neg p \equiv p|p \equiv p \downarrow p$$

$$\text{Para } C_1: p \wedge q \equiv (p|q) | (p|q) \equiv (p \downarrow p) \downarrow (q \downarrow q)$$

$$\text{Para } C_7: p \vee q \equiv (p|p) | (q|q) \equiv (p \downarrow q) \downarrow (p \downarrow q)$$

$$\text{Para } C_{13}: p \rightarrow q \equiv p | (q|q) \equiv ((p \downarrow p) \downarrow q) \downarrow ((p \downarrow p) \downarrow q)$$

$$\text{Para } C_9: p \leftrightarrow q \equiv ((p|p) | (q|q)) | (p|q) \equiv (p \downarrow (q \downarrow q)) \downarrow ((p \downarrow p) \downarrow q)$$

Estas equivalencias, que en lógica tienen un interés puramente teórico, son de utilidad en las aplicaciones a circuitos de conmutación, como veremos en el capítulo 3.

4. Modelo algebraico de la lógica de proposiciones

4.1. Algebra de Boole de las evaluaciones binarias

Suponemos que el lector tiene algunos conocimientos previos sobre estructuras algebraicas. Recuerde que un álgebra de Boole se define como una estructura

$$\langle B, +, \cdot, \bar{} \rangle$$

formada por un conjunto de base, B , con tres operaciones definidas en él, suma (+), producto (\cdot) y complementación ($\bar{}$), tales que se cumplen los siguientes axiomas:

A1: B es cerrado para las tres operaciones (es decir, si $a, b \in B$, entonces $\bar{a} \in B$, $a + b \in B$ y $a \cdot b \in B$).

A2: Existen dos elementos neutros:

$$\exists 0 \in B \text{ tal que } \forall a \in B, a + 0 = a$$

$$\exists 1 \in B \text{ tal que } \forall a \in B, a \cdot 1 = a$$

A3: Todo elemento a tiene un simétrico \bar{a} tal que

$$a + \bar{a} = 1 \text{ y } a \cdot \bar{a} = 0$$

A4: Las operaciones suma y producto son conmutativas:

$$\forall a, b \in B:$$

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

A5: Las operaciones suma y producto son asociativas:

$$\forall a, b, c \in B:$$

$$a + (b + c) = (a + b) + c$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

A6: La suma es distributiva con respecto al producto, y viceversa:

$\forall a, b, c \in B$:

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Teniendo en cuenta esta definición, se pueden demostrar teoremas o leyes (utilizando la sustitución como regla de transformación), como:

Idempotencia:

$\forall a \in B$:

$$a + a = a$$

$$a \cdot a = a$$

Absorción:

$\forall a, b \in B$:

$$a + 1 = 1$$

$$a \cdot 0 = 0$$

$$a + a \cdot b = a$$

$$a \cdot (a + b) = a$$

de Morgan:

$\forall a, b \in B$:

$$\overline{(a + b)} = (\bar{a} \cdot \bar{b})$$

$$\overline{(a \cdot b)} = \bar{a} + \bar{b}$$

etc.

Las operaciones " \neg ", " \vee " y " \wedge " definidas en el caso de evaluación binaria (Definición 3.2.1) cumplen, como es fácil comprobar, los axiomas del álgebra de Boole para el conjunto $V=\{0,1\}$, con $0=0$ y $1=1$. Por tanto,

$$\langle \{0, 1\}, \vee, \wedge, \neg \rangle$$

tiene estructura de álgebra de Boole.

Pero hay algo más interesante: la estructura de álgebra de Boole de las sentencias, o, con mayor rigor, de las clases de equivalencia entre sentencias.

4.2. Álgebra de Boole de las clases de equivalencia entre sentencias

Veamos, en primer lugar, que si $A \equiv A'$ entonces $\neg A \equiv \neg A'$. En efecto, lo primero, por definición de equivalencia, es lo mismo que decir que $(\forall i) (E_i(A) = E_i(A'))$ y, por tanto, $(\forall i) (\neg E_i(A) = \neg E_i(A'))$, de donde (por definición de la función de evaluación): $(\forall i) (E_i(\neg A) = E_i(\neg A'))$, es decir, $\neg A \equiv \neg A'$. Esto nos permite decir que si A_1, A_2, \dots están en la clase de equivalencia C_j , entonces $\neg A_1, \neg A_2, \dots$ están todas en otra clase de equivalencia, a la que llamaremos $\neg C_j$.

Del mismo modo podemos demostrar que si $A \equiv A'$ y $B \equiv B'$ entonces $(A \vee B) \equiv (A' \vee B')$. Ello nos permite decir que si A_1, A_2, \dots están todas en la clase de equivalencia C_i y B_1, B_2, \dots están todas en la clase de equivalencia C_j , entonces todas las sentencias formadas por una sentencia cualquiera del primer grupo y otra del segundo unidas por una conectiva binaria, $A_k \vee B_m$, estarán en una misma clase de equivalencia, a la que llamaremos $C_i \vee C_j$.

De este modo, hemos definido las operaciones " \neg ", " \vee ", " \wedge ", etc., en el conjunto $Q_n = S_n / \equiv$ de las clases de equivalencia de las sentencias con n variables proposicionales. (Obsérvese que ahora utilizamos los mismos símbolos con un tercer significado: operaciones en el conjunto Q_n ; los otros dos significados son los que señalábamos en la nota que acompañaba al teorema 3.1.4).

Ahora es fácil comprobar que Q_n , junto con " \neg ", " \vee " y " \wedge ", satisface los axiomas del álgebra de Boole con los elementos neutros $0 = C_0$ (clase de equivalencia correspondiente a las contradicciones) y $1 = C_N$ (clase de equivalencia correspondiente a las tautologías, en la que el subíndice N depende del número de clases de equivalencia que se formen en la partición, que, a su vez, depende del conjunto de valores semánticos, V ; en el caso de que $V=\{0,1\}$, sabemos que $N = 2^{2^n}$). Comprobémoslo para uno de los axiomas, por ejemplo, el que se refiere a la existencia de elementos simétricos (A3):

Tendremos que demostrar que, para todo C_i , se verifica que $C_i \vee \neg C_i = C_N$ y $C_i \wedge \neg C_i = C_0$. Sabemos que dada una sentencia cualquiera de un C_i cualquiera, $A \in C_i$, se cumple que $\neg(A \in \neg C_i)$, que $A \vee \neg A$ es una tautología, y, por tanto, $(A \vee \neg A) \in C_N$. También sabemos que $A \wedge \neg A$ es una contradicción y, por

tanto, $(A \wedge \neg A) \in C_0$. Pero, por otra parte, tal como se han definido las operaciones en \mathcal{Q}_n , podemos asegurar que $(A \vee \neg A) \in (C_i \vee \neg C_i)$ y que $(A \wedge \neg A) \in (C_i \wedge \neg C_i)$. Ello nos permite concluir que $C_i \vee \neg C_i = C_N$ y $C_i \wedge \neg C_i = C_0$.

Como consecuencia de la estructura de álgebra de Boole de la lógica de proposiciones, las transformaciones que estudiaremos en el capítulo 3 y que aplicaremos a los circuitos (formas canónicas, formas mínimas, etc.), son también aplicables a las sentencias de la lógica de proposiciones.

4.3. Dos teoremas del álgebra de Boole

Los dos teoremas que vamos a enunciar (omitiendo sus demostraciones, que exigen recurrir a ciertos conceptos, como relaciones de orden parcial, que ya desbordan el alcance de este libro) nos serán de utilidad en el capítulo 3.

Definición 4.3.1. Dada un álgebra de Boole $\langle B, +, \cdot, ' \rangle$, se llama *elemento atómico* a cualquier $b \in B$ ($b \neq 0$) tal que para todo $a \in B$ se cumple que o bien $a \cdot b = 0$ o bien $a \cdot b = a$.

Teorema 4.3.2. Dada un álgebra de Boole $\langle B, +, \cdot, ' \rangle$ cuyos elementos atómicos son b_1, b_2, \dots, b_n , todo $a \in B$ ($a \neq 0$) puede expresarse de manera única como suma de elementos atómicos:

$$a = b_\alpha + b_\beta + \dots + b_\zeta \quad (1 \leq \alpha, \beta, \dots, \zeta \leq n)$$

Por ejemplo, supongamos $n=2$, es decir, consideremos las sentencias que pueden formarse con dos variables proposicionales, p y q , y centrémonos en la evaluación binaria ($V=\{0,1\}$). Sabemos que en este caso hay dieciséis clases de equivalencia, cuyas evaluaciones son las dadas en el apartado 3.5. Los elementos atómicos serán aquellas clases cuyas evaluaciones sean siempre "0" excepto para una sola evaluación de p y q , es decir (véase la tabla del apartado 3.5), C_1, C_2, C_4 y C_8 . Puede comprobarse, en efecto, que el resultado del producto (conjunción) de cualquier otra clase por una de éstas es siempre ésta misma o C_0 . Y también que cualquier otra clase (salvo C_0) puede expresarse como suma (disyunción) de varios elementos atómicos: $C_{11} = C_1 + C_2 + C_8$, etc.

Definición 4.3.3. Dos álgebras de Boole, $\langle B_1, +_1, \cdot_1, '^{-1} \rangle$ y $\langle B_2, +_2, \cdot_2, '^{-2} \rangle$, son *isomorfas* si existe una aplicación biyectiva $h: B_1 \rightarrow B_2$ tal que se cumple:

$$(\forall a \in B_1) (h(a^{-1}) = h(a)^{-2})$$

$$(\forall a, b \in B_1) (h(a +_1 b) = h(a) +_2 h(b))$$

$$(\forall a, b \in B_1) (h(a \cdot_1 b) = h(a) \cdot_2 h(b))$$

Teorema 4.3.4. Si dos álgebras de Boole, B_1 y B_2 , tienen el mismo número de elementos ($\text{card}(B_1) = \text{card}(B_2)$) entonces son isomorfas.

5. Sistemas inferenciales

5.1. Análisis y generación de razonamientos

La lógica es, entre otras cosas, una herramienta para analizar los procesos de razonamiento que habitualmente se expresan en lenguaje natural. Pero dada la diversidad de matices de éste, no puede pensarse en su traducción automática al lenguaje de la lógica formal (al menos, no todavía; esto está íntimamente relacionado con uno de los campos de investigación en inteligencia artificial, el del procesamiento del lenguaje natural). Ahora bien, una vez obtenida la traducción formalizada de un determinado proceso de razonamiento, puede analizarse éste, y puede completarse con nuevas conclusiones de modo automático.

En la presentación del ejemplo 1.4.3 ya avanzábamos algo que formalizaremos en el apartado 5.2: los razonamientos pueden formalizarse como sentencias condicionales cuyo antecedente es la conjunción de las premisas y cuyo consecuente es la conclusión, y la condición necesaria y suficiente para que el razonamiento sea válido es que la sentencia así formada sea una tautología (o, equivalentemente, una tesis). Esto nos permite analizar razonamientos, pero no obtener conclusiones de modo automático. Lo que necesitamos ahora no es un sistema axiomático con el que podamos demostrar, por ejemplo, que $\vdash A$, sino un procedimiento para que, dadas P_1, P_2, \dots , podamos obtener C_1, C_2, \dots , tales que

$$\vdash (P_1 \wedge P_2 \wedge \dots \rightarrow C_1)$$

$$\vdash (P_1 \wedge P_2 \wedge \dots \rightarrow C_2)$$

Es decir, expresado en otros términos, lo que queremos no es poder demostrar teoremas a partir de unos axiomas, lo cual tiene un interés puramente teórico, sino poder derivar conclusiones a partir de unas premisas que no son tautologías, pero que sabemos (o suponemos) que son verdaderas en una determinada situación (o, lo que es lo mismo, se satisfacen para una determinada interpretación).

Pues bien, si disponemos de un repertorio de teoremas que tengan la forma $A_1 \wedge A_2 \wedge \dots \rightarrow B$, podemos pensar en el siguiente procedimiento: elegir uno tal que su antecedente se ajuste exactamente a una premisa o a la conjunción de dos o más de ellas; aplicándolo, obtenemos como conclusión el consecuente de la tesis, que se añade al conjunto de premisas, y repetir el proceso hasta que ya no puedan obtenerse más conclusiones. Esto es lo que más adelante (en el apartado 5.4) llamaremos *sistema inferencial*.

Por ejemplo, consideremos un razonamiento que se formaliza así:

$$[(p \rightarrow \neg q) \wedge (\neg q \rightarrow r)] \rightarrow (p \rightarrow r)$$

Para analizar tal razonamiento y averiguar si es correcto hemos de ver que la sentencia es un teorema, o, equivalentemente, que es una tautología. Es decir, podemos proceder de dos maneras:

- a) Tratando de demostrar la sentencia en el sistema axiomático. (En este caso, basta aplicar la sustitución $(\neg q/q)$ en la primera de las leyes de transitividad enunciadas en el apartado 2.5.6).
- b) Construyendo la tabla de verdad, para ver si la sentencia es una tautología, que es lo que hacíamos en los ejemplos del apartado 1.4. Esta forma podría parecer, en principio, más fácil, pero obsérvese que, por ejemplo, el teorema

$$[(p_1 \rightarrow p_2) \wedge (p_2 \rightarrow p_3) \wedge \dots \wedge (p_9 \rightarrow p_{10})] \rightarrow (p_1 \rightarrow p_{10})$$

requeriría una tabla de $2^{10} = 1024$ líneas (una para cada una de las distintas evaluaciones del conjunto de variables proposicionales).

Pero lo que más nos interesa no es el análisis del razonamiento, sino su generación. En este caso, dadas las premisas

$$P_1: p \rightarrow \neg q$$

$$P_2: \neg q \rightarrow r$$

qué conclusión (o conclusiones) podemos obtener? Pues bien, dado que la sentencia anteriormente escrita es un teorema en forma de condicional que tiene como antecedente la conjunción de P_1 y P_2 , podemos afirmar como conclusión el consecuente:

$$C: p \rightarrow r$$

Esto se llama *inferencia*: a partir de las premisas P_1 y P_2 hemos inferido la conclusión C . ¿Se pueden inferir otras conclusiones? ¿Qué conclusiones son válidas? Toda C_i tal que $\vdash (P_1 \wedge P_2 \rightarrow C_i)$ será una conclusión válida. Pasemos ahora a precisar estas ideas.

5.2. Implicación lógica y razonamientos deductivos

Definición 5.2.1. Un conjunto de sentencias $\{A_1, A_2, \dots, A_n\}$ *implica lógicamente* a una sentencia A si (y sólo si) toda interpretación que satisface al conjunto (es decir, que satisface a todas y cada una de las sentencias del conjunto) satisface también a A . Para indicar que $\{A_1, A_2, \dots, A_n\}$ implica lógicamente a A escribiremos:

$$\{A_1, A_2, \dots, A_n\} \models A$$

Obsérvese que la equivalencia entre sentencias, definida en el apartado 3.5, puede expresarse por medio de este concepto de implicación: A y B son equivalentes si (y sólo si) $A \models B$ y $B \models A$.

Teorema 5.2.2. En un sistema axiomático consistente, $\{A_1, A_2, \dots, A_n\} \models A$ si (y sólo si)

$$\vdash (A_1 \wedge A_2 \wedge \dots \wedge A_n \rightarrow A)$$

La demostración, que dejamos al cuidado del lector, es similar a la del teorema 3.5.2.

Definición 5.2.3. Un *razonamiento* es un par $\langle Pr, C \rangle$, donde $Pr = \{P_1, P_2, \dots, P_n\}$, es un conjunto finito de sentencias llamadas *premisas* y C es una sentencia llamada *conclusión*.

Definición 5.2.4. Un *razonamiento válido*, o *razonamiento deductivo*, es un razonamiento $\langle Pr, C \rangle$ tal que $Pr \models C$. Aplicando el teorema 5.3.2, la condición necesaria y suficiente para que un razonamiento sea válido es:

$$\vdash (P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow C)$$

5.3. Reglas de inferencia

El número de razonamientos deductivos, es decir, de conjuntos de premisas, Pr y de conclusiones, C , tales que $Pr \models C$, es infinito, como infinito es el número de leyes de la forma

$$\vdash (P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow C)$$

Cuando razonamos a partir de un conjunto de premisas, normalmente no lo hacemos en un solo paso. Más bien, escogemos dos de las premisas, aplicamos un "razonamiento elemental" para obtener una conclusión intermedia, añadimos esta conclusión al conjunto de premisas, volvemos a elegir otras dos premisas, y así sucesivamente.

Estos "razonamientos elementales" están basados en "reglas de inferencia". Una *regla de inferencia* es la declaración de las condiciones bajo las cuales puede obtenerse una conclusión a partir de unas premisas (normalmente, una o dos). A toda tesis (o ley) del cálculo proposicional que tenga la forma $A_1 \wedge A_2 \wedge \dots \rightarrow C$ puede hacérsele corresponder una regla de inferencia.

Pero *ley* y *regla* no son la misma cosa. La diferencia es lingüística: la ley pertenece al *lenguaje* del cálculo, y *representa* a una regla o esquema válido de inferencia; la regla, entonces, pertenece al *metalenguaje* del cálculo.

Por ejemplo, la ley de *modus ponens* (apartado 2.5.6) es:

$$[p \wedge (p \rightarrow q)] \rightarrow q$$

o bien, si A y B son sentencias, utilizando la sustitución $\{A/p, B/q\}$,

$$[A \wedge (A \rightarrow B)] \rightarrow B$$

La correspondiente regla de inferencia se expresaría así:

'De A y de $A \rightarrow B$ puede inferirse B '.

'Puede inferirse' pertenece al metalenguaje del cálculo y establece una relación de deducibilidad que sería abusivo representar por " \rightarrow ", porque este último símbolo pertenece al lenguaje. Por ello, la forma habitual de simbolizar esa regla de inferencia es la siguiente:

$$\frac{A \quad A \rightarrow B}{B}$$

Nótese que, aun guardando cierto parecido, esta regla de inferencia no es exactamente igual que la regla de transformación RT3 (regla de separación) definida en el apartado 2.5.4. En efecto, esta última dice: 'si A y $A \rightarrow B$ son tesis (o sea, *siempre* son verdaderas), B también lo es', mientras que la regla de inferencia correspondiente al *modus ponens* dice: 'en el caso de que tanto A como $A \rightarrow B$ sean verdaderas, B también lo es'. La generalización para escribir cualquier regla de inferencia es obvia: cada condición se escribe en una línea, y la conclusión en una final, bajo una raya. El lector no tendrá dificultades para escribir en esta forma, por ejemplo, las reglas correspondientes a las leyes de *modus tollens*, de transitividad, de inferencia de la alternativa, del silogismo disyuntivo y del dilema constructivo, dadas en el apartado 2.5.6. Dos reglas de inferencia sencillas pero muy utilizadas, que se basan en leyes triviales son:

- la regla de introducción de conjunción:

$$\frac{A \quad B}{A \wedge B}$$

- la regla de eliminación de conjunción:

$$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B}$$

Para ilustrar lo estudiado en este apartado, volvamos al ejemplo 1.4.3. Teníamos las premisas:

$$P1: c \rightarrow s$$

$$P2: s \leftrightarrow \neg m$$

$P2$ puede descomponerse en dos:

$$P2a: s \rightarrow \neg m; P2b: \neg m \rightarrow s$$

En efecto, $(s \leftrightarrow \neg m) \leftrightarrow (s \rightarrow \neg m) \wedge (\neg m \rightarrow s)$ es un teorema, y, por tanto, $P2$ y $P2a \wedge P2b$ son equivalentes, de acuerdo con el teorema (meta-teorema) 3.5.2.

Según las leyes de transitividad (apartado 2.5.6), una regla de inferencia es:

$$\frac{\begin{array}{c} A \rightarrow B \\ B \rightarrow C \end{array}}{A \rightarrow C}$$

Aplicándola a las premisas $P1$ y $P2a$ se infiere la conclusión:

$$C: c \rightarrow \neg m$$

Lo que hacíamos en la exposición del ejemplo 1.4.3 era ver que la sentencia $(P_1 \wedge P_2) \rightarrow C$ es una tautología. La consistencia del sistema axiomático nos permite, de esta forma, asegurar que esta sentencia es también una ley. De aquí, aplicando el teorema 5.2.2, podemos decir, entonces, que $P1$ y $P2$ implican C , y, así siendo, finalmente, por la definición 5.2.4 concluimos que el razonamiento es válido.

¿Es éste el único razonamiento válido con esas premisas? No, porque, según las definiciones 5.2.1 y 5.2.4, basta con que una sentencia se satisfaga para todas las evaluaciones que satisfacen a las premisas para que tal sentencia sea una conclusión de tales premisas, siendo indiferente lo que ocurra con las otras evaluaciones. Rehaciendo la tabla de verdad, podemos poner en ella diversas conclusiones válidas:

c	s	m	P_1	P_2	$P_1 \wedge P_2$		C	C'	C''	C'''
0	0	0	1	0	0	1	1	0	1	1
0	0	1	1	1	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1	1
0	1	1	1	0	0	1	1	1	1	1
1	0	0	0	0	0	1	0	0	0	1
1	0	1	0	1	0	0	0	0	1	0
1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0	1	0

$$C: c \rightarrow \neg m$$

$$C': c \rightarrow s \wedge \neg m$$

$$C'': \neg s \rightarrow m$$

$$C''': m \rightarrow \neg c$$

etcétera.

Y, en general, cualquier sentencia que se satisfaga para i_1, i_2 e i_6 es una conclusión de un razonamiento válido. (Entre ellas habrá muchas conclusiones triviales, como las propias premisas, la conjunción de las mismas, etc., y, desde luego, cualquier tautología: según la definición de "razonamiento válido", toda tautología puede considerarse como "conclusión válida" de cualquier conjunto de premisas).

Pero C', C'', C''' , etc. también pueden obtenerse por inferencia: C' se infiere de P_1 y de C (por la regla de inferencia que corresponde al teorema $(p \rightarrow q) \wedge (p \rightarrow r) \rightarrow (p \rightarrow q \wedge r)$); C'' , de P_2a (teorema: $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$); C''' , de C , etc.

5.4. Sistemas inferenciales

En los tratados de lógica se presentan conjuntos seleccionados de reglas de inferencia bajo el nombre de "sistemas de deducción natural". El problema para su aplicación en el diseño de un sistema automático (es decir, un programa para ordenador que ejecute inferencias) es que, dado un problema, es preciso tener una cierta "habilidad" o "conocimiento" (que se expresaría en forma de metarreglas) para saber en qué orden aplicar las reglas y a qué premisas. Más adelante veremos una formulación que permite trabajar con una sola regla de inferencia. En cualquier caso, el conjunto de reglas que se utilice debe ser *consistente*, y, a ser posible, *completo*. (Por razones prácticas, en los sistemas pensados para programarse en ordenador a veces se sacrifica la completitud en aras de la eficacia). Estos conceptos son paralelos a los definidos para un sistema axiomático: Así como un sistema axiomático es completo si toda tautología es una tesis, un sistema inferencial lo es si toda conclusión de un razonamiento correcto puede inferirse; y así como un sistema axiomático es consistente si toda tesis es una tautología, un sistema inferencial lo es si toda inferencia corresponde a un razonamiento válido. Vamos a precisarlo con algo más de detalle.

Definición 5.4.1. Un *proceso inferencial (monótono)* es una secuencia dinámica de estados, en la que el estado inicial es un conjunto de premisas, y cada estado se obtiene del anterior añadiéndole la conclusión obtenida aplicando una regla de inferencia.

Definición 5.4.2. Un *sistema inferencial* es un conjunto de reglas de inferencia junto con unas metarreglas que especifican cómo se aplican las reglas. Estas metarreglas constituyen la *estrategia* del sistema: orden de aplicación de las reglas, orden de elección de las premisas, orden en que se añaden las conclusiones a las premisas para obtener nuevas conclusiones, etc.

La ejecución de un sistema inferencial sobre un conjunto de premisas que forman el estado inicial da lugar a un proceso inferencial. En un estado cualquiera de éste, la estrategia permite decidir cuál de las reglas se aplica a cuáles de las sentencias que forman el estado para obtener una conclusión que se añade al estado actual, lo que genera el estado siguiente.

Teorema 5.4.3. Todo sistema inferencial cuyas reglas de inferencia se puedan formalizar como tesis de un sistema axiomático consistente es un sistema inferencial consistente.

Demostración:

Sean P_1, P_2, \dots, P_n las premisas y C la conclusión de una regla de inferencia cualquiera. La hipótesis del teorema establece que

$$\vdash [(P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow C)]$$

Como el sistema axiomático es consistente, el teorema 5.2.2 nos permite asegurar que

$$\{P_1 \wedge P_2 \wedge \dots \wedge P_n\} \models C$$

Y por la definición 5.2.4, el razonamiento $\langle \{P_1 \wedge P_2 \wedge \dots \wedge P_n\}, C \rangle$ es un razonamiento válido. Como todas las inferencias se obtienen por aplicación de las reglas de inferencia, y lo anterior es aplicable a cualquier regla de inferencia, concluimos que toda sentencia que pueda inferir el sistema constituye, junto con las premisas, un razonamiento válido.

Así pues, la consistencia de un sistema inferencial está asegurada si las reglas de inferencia que utilizamos son "buenas". No así la completitud. Esta depende de la estrategia, y, como decíamos al principio de este apartado, a veces se adopta una estrategia sencilla pero que no da lugar a un sistema completo. Por ejemplo, en muchos "sistemas expertos" (que comentaremos en el capítulo 6) se utiliza un sistema inferencial muy sencillo que tiene como única regla de inferencia la basada en la ley de *modus ponens*. Tal sistema no es completo, pero sí suficiente para los problemas a los que se dirige.

5.5. Forma clausulada de la lógica de proposiciones

La regla de inferencia que estudiaremos en el apartado 5.7 se aplica únicamente a una forma especial de sentencias, pero vamos a ver que toda sentencia de la lógica de proposiciones puede expresarse de modo equivalente en esa forma.

Definición 5.5.1. Una *cláusula* es una sentencia de la forma:

$$l_1 \vee l_2 \vee \dots \vee l_n$$

Es decir, una cláusula es una disyunción de literales (recuérdese que un "literal" es una variable proposicional sola o con la negación). Por ejemplo: $\neg p \vee q$, $p \vee q \vee r$, $p \vee \neg q \vee r \vee \neg s$, son cláusulas.

Definición 5.5.2. Diremos que una sentencia está en *forma clausulada* si tiene la forma:

$$(l_{11} \vee l_{12} \vee \dots) \wedge (l_{21} \vee l_{22} \vee \dots) \wedge \dots$$

Es decir, una sentencia en forma clausulada es una conjunción de cláusulas.

Dado que las operaciones de conjunción y disyunción son asociativas y conmutativas (esta afirmación está basada en teoremas del cálculo, que hemos mencionado sin demostrarlos en el apartado 2.3), podemos decir que:

- a) una cláusula es una colección de literales (implícitamente unidos por disyunciones), y
- b) una sentencia en forma clausulada es una colección de cláusulas (implícitamente unidas por conjunciones).

Teorema 5.5.3. Para toda sentencia de la lógica de proposiciones existe una sentencia equivalente en forma clausulada.

Demostración:

Daremos una demostración constructiva, es decir un procedimiento que permite transformar cualquier sentencia en otra equivalente que está en forma clausulada. Supondremos que en la sentencia original no se utilizan más que las cuatro conectivas binarias más conocidas. (Ya sabemos que cualquiera otra puede expresarse en función de una de ellas y de la negación). El procedimiento, que, naturalmente, se basa en teoremas de tipo equivalencia, consta de tres pasos:

1. Eliminación de condicionales y bicondicionales mediante las equivalencias derivadas de la definición de ambos:

$$(A \rightarrow B) \equiv (\neg A \vee B)$$

$$(A \leftrightarrow B) \equiv (\neg A \vee B) \wedge (A \vee \neg B)$$

2. Introducción de negaciones de modo que queden afectando sólo a variables proposicionales, mediante aplicación sucesiva de las equivalencias derivadas de las leyes de de Morgan:

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

así como de la ley de doble negación:

$$\neg\neg A \equiv A$$

Con ello, se habrá llegado a una sentencia formada por literales unidos por las conectivas " \vee " y " \wedge ".

3. Paso a forma clausulada, distribuyendo " \wedge " sobre " \vee " mediante la equivalencia:

$$[(A_1 \wedge A_2) \vee A_3] \equiv [(A_1 \vee A_3) \wedge (A_2 \vee A_3)]$$

Con frecuencia, la forma obtenida puede simplificarse. Así, si dentro de una cláusula aparece dos o más veces el mismo literal, se escribe una sola vez ($l \vee l \vee l \vee \dots \leftrightarrow l$); si aparece un literal y su complementario ($p \vee \neg p$), la cláusula es una tautología, y puede eliminarse del conjunto de cláusulas; si hay dos o más cláusulas idénticas, se escribe una sola de ellas.

Por ejemplo, pasemos a forma clausulada la sentencia:

$$\neg \{ [(p \wedge q) \rightarrow p] \rightarrow [(q \vee r) \wedge (\neg q \wedge \neg r)] \}$$

$$1. \neg \{ \neg [\neg(p \wedge q) \vee p] \vee [(q \vee r) \wedge (\neg q \wedge \neg r)] \}$$

$$2. [\neg(p \wedge q) \vee p] \wedge \neg[(q \vee r) \wedge (\neg q \wedge \neg r)]$$

$$[(\neg p \vee \neg q) \vee p] \wedge [\neg(q \vee r) \vee \neg(\neg q \wedge \neg r)]$$

$$[\neg p \vee \neg q \vee p] \wedge [(\neg q \wedge \neg r) \vee (q \vee r)]$$

$$3. [\neg p \vee \neg q \vee p] \wedge [(\neg q \vee q \vee r)] \wedge [\neg r \vee q \vee r]$$

En este ejemplo, cada una de las tres cláusulas obtenidas es una tautología, indicando así que la sentencia original era un teorema. De acuerdo con las reglas de simplificación, las tres cláusulas pueden hacerse desaparecer, pero, para no confundir con la cláusula vacía, que, como veremos luego, corresponde a todo lo

contrario (una contradicción), representaríamos el resultado final como la única cláusula $\neg p \vee p$.

Definición 5.5.4. Diremos que trabajamos en la *forma clausulada de la lógica* cuando expresamos todas las sentencias en forma clausulada.

La forma clausulada es más concisa (aunque menos natural) que la forma "estándar" de la lógica. Por ejemplo, las seis sentencias

$$p \wedge q \rightarrow r; p \wedge \neg r \rightarrow \neg q; q \wedge \neg r \rightarrow \neg p;$$

$$p \rightarrow \neg q \vee r; q \rightarrow \neg p \vee r; \neg r \rightarrow \neg p \vee \neg q;$$

se escriben en forma clausulada de la misma manera:

$$\neg p \vee \neg q \vee r$$

Sin embargo, aunque este ejemplo pueda inducir a pensar lo contrario, la forma clausulada no es única: pueden existir sentencias que estén en formas clausuladas diferentes y que sean equivalentes. Para ilustrarlo, consideremos esta sentencia:

$$(p \leftrightarrow q) \wedge \neg(p \wedge q \wedge r)$$

La aplicación del procedimiento anterior nos conduce a la forma clausulada:

$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q \vee \neg r)$$

Y, como puede comprobarse fácilmente, estas otras dos formas clausuladas son equivalentes a ella:

$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg r)$$

$$(\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg q \vee \neg r)$$

5.6. Las cláusulas como sentencias condicionales

Dada una cláusula cualquiera, $l_1 \vee l_2 \vee \dots$, podemos hacer otras transformaciones, basadas también en equivalencias, que nos permiten escribirla como una sentencia condicional. Para ello, escribimos primero los literales negativos y luego los positivos:

$$\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_k \vee q_1 \vee q_2 \vee \dots \vee q_m$$

Por la generalización de una de las leyes de de Morgan, esta sentencia es equivalente a:

$$\neg(p_1 \wedge p_2 \wedge \dots \wedge p_k) \vee q_1 \vee q_2 \vee \dots \vee q_m$$

Y por la ley $(\neg A \vee B) \leftrightarrow (A \rightarrow B)$, esta otra sentencia es equivalente a las anteriores:

$$p_1 \wedge p_2 \wedge \dots \wedge p_k \rightarrow q_1 \vee q_2 \vee \dots \vee q_m$$

Con esta nueva escritura vemos que la lectura metalógica de una cláusula es: "si se dan como antecedentes todos los literales negativos, entonces se siguen como consecuentes uno o varios de los literales positivos".

Hay algunos casos particulares de cláusulas que tienen un interés especial en las aplicaciones:

- a) *Cláusulas de Horn con cabeza*. Son las que solamente tienen un literal positivo:

$$(\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_k \vee q) \equiv (p_1 \wedge p_2 \wedge \dots \wedge p_k \rightarrow q)$$

- b) *Cláusulas de Horn sin cabeza*. Son las que no tienen ningún literal positivo:

$$(\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_k) \equiv \neg(p_1 \wedge p_2 \wedge \dots \wedge p_k)$$

En este caso, las variables proposicionales p_1, p_2, \dots, p_k son *incompatibles*, es decir no es posible que sean todas verdaderas.

- c) Las que no tienen ningún literal negativo:

$$(q_1 \vee q_2 \vee \dots \vee q_m)$$

En este caso, al menos una de las variables es verdadera.

- d) La *cláusula vacía*, φ , en la que han desaparecido todos los literales, y que, como veremos, aparece en un proceso inferencial cuando las premisas son incompatibles.

5.7. La regla de resolución

Definición 5.7.1. La regla de inferencia llamada *resolución* se aplica a dos premisas en forma de cláusulas, tales que tengan en común un literal positivo en una y negativo en otra, y a las que llamaremos *generatrices*. La inferencia consiste en construir otra cláusula, llamada *resolvente*, formada por la disyunción de todos los literales de las generatrices salvo el común.

Por ejemplo:

$$\frac{p \vee \neg q \quad \neg p \vee r \vee s}{\neg q \vee r \vee s}$$

Como toda regla de inferencia, la resolución se fundamenta en una tesis, concretamente, en

$$\vdash [(\neg p \vee A) \wedge (p \vee B) \rightarrow (A \vee B)]$$

que es una generalización de las leyes de inferencia de la alternativa. Por consiguiente, de acuerdo con el teorema 5.4.5, todo sistema inferencial basado en la resolución será consistente.

Las reglas de inferencia "clásicas" pueden expresarse como resoluciones, si previamente se escriben las premisas en forma clausulada. Por ejemplo:

- *modus ponens*:

$$\frac{A \quad A \rightarrow B}{B}$$

resolución:

$$\frac{A \quad \neg A \vee B}{B}$$

- *modus tollens*:

$$\frac{\neg B \quad A \rightarrow B}{\neg A}$$

resolución:

$$\frac{\neg B \quad \neg A \vee B}{\neg A}$$

- *transitividad*:

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C}$$

resolución:

$$\frac{\neg A \vee B \quad \neg B \vee C}{\neg A \vee C}$$

Para completar el sistema inferencial tenemos que dar una estrategia sobre la que basar el procedimiento de aplicación de la resolución. El más inmediato es el de la búsqueda exhaustiva:

Definición 5.7.2. Dadas n premisas en forma de cláusulas, la *búsqueda exhaustiva* consiste en aplicar la resolución a todas las parejas posibles de cláusulas, añadir las resolventes al conjunto de cláusulas, aplicar la resolución a todas las nuevas parejas, y así sucesivamente, hasta que en la aplicación de la resolución no se obtengan nuevas resolventes.

Por lo dicho anteriormente (teorema 5.4.3), el sistema es consistente, pero ¿será completo? En principio, parece que no lo fuese, porque, como sabemos, para dos premisas puede haber más de una conclusión, mientras que la resolución sólo produce una (la resolvente). Ahora bien, si aplicamos repetidamente la resolución (añadiendo toda conclusión al conjunto de premisas) y considerando como inferencias no solamente las resolventes que se van obteniendo, sino también todas las conjunciones en el conjunto de premisas y resolventes, todas las cláusulas que se obtienen por disyunción de dos o más cláusulas en ese conjunto y todas las conclusiones "triviales" (entendiendo como tales las disyunciones de cualquier premisa o conclusión con cualquier literal; por ejemplo, de la premisa " p " son conclusiones triviales " $p \vee q$ " y " $p \vee \neg q$ "), entonces puede demostrarse que se infieren todas las conclusiones posibles. Veámoslo (sin entrar en la demostración) con algunos ejemplos.

Ejemplo 5.7.3. Consideremos de nuevo la sentencia estudiada al final del apartado 5.5, y supongamos que corresponde a dos premisas:

$$P1: (p \leftrightarrow q)$$

$$P2: \neg(p \wedge q \wedge r)$$

En forma clausulada:

$$P1a: (\neg p \vee q)$$

$$P2b: (p \vee \neg q)$$

$$P2: (\neg p \vee \neg q \vee \neg r)$$

Si aplicamos la resolución a $P1a$ y a $P2$ obtenemos la resolvente:

$$C1: (\neg p \vee \neg r)$$

Y aplicándola a $P1b$ y $P2$,

$$C2: (\neg q \vee \neg r)$$

Resolviendo con todas las parejas que pueden formarse con $C1$, $C2$ y las premisas, se puede comprobar que no se obtienen más conclusiones. (Por ejemplo, la resolución de $C1$ con $P1b$ conduce de nuevo a $C2$). Si se analizan todas las conclusiones posibles (construyendo la tabla de verdad) se comprobará que todas ellas corresponden a algún elemento del conjunto formado por estas cinco cláusulas ($P1a$, $P1b$, $P2$, $C1$, $C2$) más las conclusiones "triviales" (por ejemplo, de $P1a$ son conclusiones triviales $\neg(p \vee q \vee r)$ y $\neg p \vee q \vee \neg r$) o a una cláusula formada por la disyunción de dos o más de ellas, o a una conjunción de dos o más de ellas.

Ejemplo 5.7.4. En el apartado 5.3 veíamos algunas conclusiones posibles de unas premisas enunciadas en el ejemplo 1.4.3. Las premisas en forma clausulada son:

$$P1: \neg c \vee s$$

$$P2a: \neg s \vee \neg m$$

$$P2b: m \vee s$$

Resolviendo $P1$ y $P2a$,

$$C: \neg c \vee \neg m$$

De $P2a$ y $P2b$ se obtienen $\neg s \vee s$, o $\neg m \vee m$, que son tautologías, y de $P2b$ y C , $\neg c \vee s$, que es $P1$. Y, como puede comprobarse, todas las conclusiones posibles equivalen a conjunciones o disyunciones en el conjunto $\{P1, P2a, P2b, C\}$. Así, las señaladas en el apartado 5.3:

C' : $c \rightarrow s \wedge \neg m$ es, en forma clausulada, $(\neg c \vee s) \wedge (\neg c \vee \neg m)$, conjunción de $P1$ y C

C'' : $\neg s \rightarrow m$ es en forma clausulada igual que $P2a$.

C''' : $m \rightarrow \neg c$ es en forma clausulada igual que C

Ejemplo 5.7.5. Apliquemos ahora la resolución al ejemplo 1.4.4. Las premisas son:

$$P1: \neg j \vee \neg c$$

$$P2: j \vee \neg d$$

$$P3: \neg s \vee c$$

De $P1$ y $P2$, $C1$: $\neg c \vee \neg d$

De $P1$ y $P3$, $C2$: $\neg j \vee \neg s$

De $P2$ y $C2$, $C3$: $\neg d \vee \neg s$

El lector puede comprobar que no hay más conclusiones (salvo disyunciones y conjunciones de las premisas y las tres conclusiones obtenidas). Puede, asimismo, analizar, las evaluaciones en forma de condicional de estas conclusiones. La que se veía en el ejemplo 1.4.4, $s \rightarrow \neg d$, es, puesta en forma clausulada, $C3$.

5.8. Refutación

La *refutación* es un procedimiento útil cuando lo que se pretende no es generar cuantas conclusiones sean posibles, sino comprobar si una determinada conclusión es válida o no.

Definición 5.8.1. La *refutación* consiste en comprobar que el conjunto de cláusulas formado por las correspondientes a las premisas y la que procede de la conclusión negada es una contradicción, lo cual demuestra que la conclusión se infiere de las premisas.

El fundamento de la refutación es la "ley de reducción al absurdo" (apartado 2.5.6). Si en ese teorema hacemos la sustitución $\{p/(P \rightarrow C)\}$, donde $P = P1 \wedge P2 \wedge \dots \wedge Pn$ es la conjunción de premisas y C la conclusión a comprobar, resulta:

$$\vdash \{ [\neg(P \rightarrow C) \rightarrow (q \wedge \neg q)] \leftrightarrow (P \rightarrow C) \}$$

o, lo que es lo mismo,

$$\vdash \{ [(P \wedge \neg C) \rightarrow (q \wedge \neg q)] \leftrightarrow (P \rightarrow C) \}$$

Es decir, " $(P \rightarrow C)$ es verdadera (y, por tanto, C es una conclusión válida) si (y sólo si) de la conjunción de P y $\neg C$ resulta una contradicción".

Comprobemos que en el último ejemplo del apartado anterior puede inferirse $s \rightarrow \neg d$. Pasemos primero su negación a forma clausulada:

$$\neg(s \rightarrow \neg d) \equiv \neg(\neg s \vee \neg d) \equiv s \wedge d$$

(es decir, resultan dos cláusulas: s y d).

Apliquemos repetidamente la resolución a $\{P1, P2, P3, s, d\}$:

La *semántica* permite establecer la correspondencia entre las construcciones simbólicas del cálculo y los elementos de la realidad que pretendemos representar con ellas. En el apartado 3 hemos definido los conceptos de *interpretación* (asignación de elementos de la realidad a variables proposicionales y sentencias) y evaluación (asignación, para una determinada interpretación, de valores de verdad o falsedad a las variables proposicionales y sentencias).

Así, la evaluación de una sentencia como verdadera o falsa depende de la interpretación, pero hemos visto que ciertas sentencias son *tautologías*: su evaluación es verdadera para todas las interpretaciones posibles de las variables proposicionales que la forman. El sistema axiomático es *completo* si toda tautología es una tesis ($\vdash A \rightarrow \vdash A$), y es *consistente* si toda tesis es una tautología ($\vdash A \rightarrow \vdash A$)⁵

Hemos dedicado el apartado 4 a la estructura de álgebra de Boole de la lógica de proposiciones, modelo que utilizaremos en el capítulo 3.

Finalmente, en el apartado 5 nos hemos centrado en los *razonamientos válidos* (o *deductivos*) y en los *sistemas inferenciales*. Aquí ya no consideramos "axiomas" (en el sentido que se le da a esta palabra en un sistema axiomático), sino *premisas*, que son sentencias supuestamente verdaderas en un determinado contexto, pero que no tienen por qué serlo siempre (algunos textos de lógica aplicada llaman "axiomas" a estas premisas). Una premisa puede ser, por ejemplo, una simple variable proposicional. La idea es que, *supuesto* que las premisas sean verdaderas, el sistema pueda inferir de ello *conclusiones*. El diagrama ahora podría ser este otro:

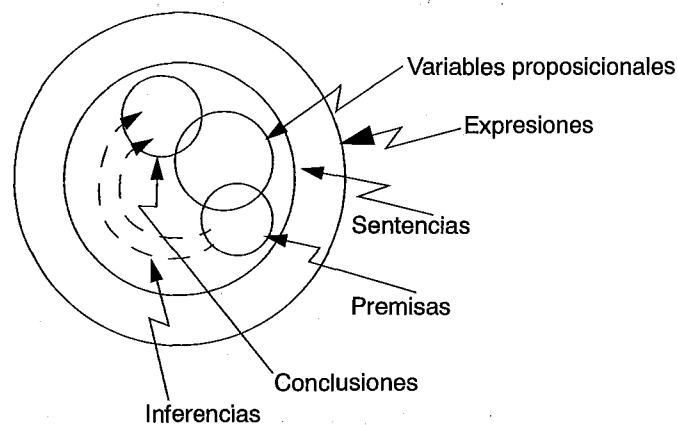


Figura 2.2.

⁵ Las dos expresiones que acabamos de escribir entre paréntesis son sentencias de la lógica de predicados. En efecto, " \vdash " y " \vdash " son predicados que se aplican a la sentencia A .

Hemos definido el concepto de *razonamiento válido*, o *deductivo*, como aquél formado por un conjunto de premisas, $\{P_1, P_2, \dots, P_n\}$, y una conclusión, C , tales que las premisas implican lógicamente a la conclusión. Y hemos visto que la condición necesaria y suficiente para que el razonamiento sea válido es que la sentencia $(P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow C)$ sea una tesis.

Después hemos definido los conceptos de *regla de inferencia*, *proceso inferencial* y *sistema inferencial*, así como de *consistencia* y *completitud* de un sistema inferencial. Un sistema inferencial es *completo* si, para cualquier conjunto de premisas, el sistema es capaz de inferir todas las conclusiones que junto con las premisas constituyen razonamientos válidos, y es *consistente* si, para cualquier conjunto de premisas, toda conclusión que llegue a inferir el sistema es tal que junto con las premisas constituye un razonamiento válido.

Por último, hemos presentado la "resolución". En la *forma clausulada de la lógica* todas las sentencias se expresan como colecciones (conjunciones) de *cláusulas*, siendo éstas colecciones (disyunciones) de *literales*. La *resolución* es una regla de inferencia que se aplica a dos cláusulas (*generatrices*) y, si tienen una pareja de literales complementarios, produce como conclusión otra cláusula (*resolvente*). Esta regla nos permite realizar de manera única inferencias que en los tratados de lógica requieren distintas reglas.

Concluiremos este capítulo con tres *consideraciones importantes* que tienen que ver con la aplicación a la informática de los conceptos estudiados:

- Ya en el primer apartado del primer capítulo hacíamos observar que el término "razonamiento" tiene dos significados: uno *funcional* (la relación entre unas premisas y una conclusión) y otro *procesal* (la actividad del "agente", sea éste una persona o una máquina, que razona). Esto no es simple retórica. Nos interesa la lógica porque buscamos modelos para implementar el razonamiento en los ordenadores. Pero la lógica sólo se ocupa de los aspectos funcionales. Los tratados de lógica nos presentan, como decíamos en el apartado 5.4, "sistemas de deducción natural", que no son más que conjuntos de reglas de inferencia que el lector puede aplicar para razonar formalmente; a este lector se le supone dotado de ciertas habilidades misteriosas (la "estrategia") que le permiten decidir la secuencia de aplicación de las reglas a unas premisas concretas. ¿Y dónde podemos encontrar modelos para los aspectos procesales? Pues bien, la ciencia que se ocupa de este asunto es la *psicología cognoscitiva*. Esta observación nos conduce a una de las actividades de trabajo fundamentales en el campo de la informática conocido como "inteligencia artificial": la aplicación de modelos de la psicología (como redes semánticas, o marcos, que estudiaremos en el apartado 4 del capítulo 6) a la implementación en ordenador de sistemas inferenciales. La estrategia esbozada en el apartado 5.7 (la "búsqueda exhaustiva") es un modelo trivial. El lector interesado en estudiar este asunto puede acudir a la bibliografía que citamos en el apartado 6 del capítulo 6.

- Ya hemos comentado, en el apartado 3.2, otra de las limitaciones de la lógica "clásica": la de ser "binaria", y, por tanto, no poder representar conocimientos con incertidumbre, ni razonar con ellos. También hemos dicho que en el capítulo 5 estudiaremos algunos enfoques lógicos para este problema. Pero hay otra limitación, que no hemos mencionado: la definición de "proceso inferencial" dada en el apartado 5.4 (definición 5.4.1) corresponde a lo que se llama *razonamiento monótono*. En un razonamiento de este tipo, las conclusiones que se van obteniendo son siempre "definitivas". Pero la mente humana opera muchas veces de manera no monótona: obtiene conclusiones "provisionales", en el sentido de que pueden verse invalidadas a la luz de nuevas evidencias. En inteligencia artificial se estudian, para ciertas aplicaciones, procesos inferenciales con razonamiento no monótono. Haremos referencia a la lógica no monótona en el apartado 2.4 del capítulo 5.
- También hemos comentado varias veces lo limitado que es el lenguaje de la lógica de proposiciones. Tanto, que prácticamente no hay ninguna aplicación real que se limite a ella. Por eso, la mayoría de los textos de lógica aplicada a la informática presentan directamente la *lógica de predicados*. Aquí, sin embargo, hemos preferido empezar por la de proposiciones por razones didácticas: las ideas básicas (razonamientos, sistemas inferenciales, completitud, consistencia, resolución, etc.) se entienden mucho mejor en este marco simplificado, y conceptualmente son las mismas que en lógica de predicados, como veremos en el capítulo 4.

7. Notas histórica y bibliográfica

Es bien sabido que la lógica, como ciencia del análisis del comportamiento racional, tiene una historia milenaria, y que fue, sobre todo, Aristóteles quien sentó las bases de los desarrollos posteriores. De la lógica aristotélica se puede decir que era formal (atendía a la forma de los razonamientos), pero no formalizada simbólicamente. Algunos filósofos medievales, entre los que cabe destacar al mallorquín Ramon Llull, hicieron algunos avances hacia la formalización de la lógica, pero los trabajos más importantes en este sentido no se realizaron hasta la segunda mitad del siglo pasado, cuando Boole (1854) elaboró su modelo algebraico de la lógica de proposiciones y Frege (1884) formalizó la de predicados, y la primera de éste, cuando aparece, como obra culminante, el libro de Whitehead y Russell (1910-1913).

Las conectivas de "negación alternativa" y "negación conjunta" fueron propuestas por Sheffer en 1913, y en lógica tienen un interés puramente teórico, y casi anecdótico. Sin embargo, como veremos en el próximo capítulo, corresponden a funciones de conmutación tecnológicamente importantes (que se suelen llamar "NAND" y "NOR", respectivamente, en la literatura técnica).

En los años 60 se desarrollaron diversos métodos orientados al procesamiento automático de las inferencias, de los que el más conocido es el de "resolución", debido a Robinson (1965). Lo que aquí hemos presentado es la particularización a la lógica de proposiciones de ese método, que en el capítulo 4 estudiaremos ya en su integridad.

Hay libros de introducción a la lógica cuya lectura, además de recomendable, es muy amena. Destacamos el de Ferrater y Leblanc (1962) (donde se llama "lógica sentencial" a lo que aquí hemos denominado "lógica de proposiciones") y el de Deaño (1986) (donde se le llama "lógica de enunciados"). El ejemplo 1.4.3 está tomado del primero, y el 1.4.4, del segundo (Deaño hace uso de este ejemplo, procedente de Lewis Carroll, para ilustrar la necesidad de la lógica de predicados; aquí hemos visto que también se puede formalizar en lógica de proposiciones, aunque abusando un poco del lenguaje natural). En el libro de Deaño nos hemos inspirado también para la explicación sobre el significado del condicional y la diferencia entre "leyes" y "reglas de inferencia". El ejemplo 1.4.5 es una adaptación de otro similar de Gilbert (1976), un libro sobre álgebra aplicada cuyo capítulo 2 se dedica a la lógica y los circuitos lógicos.

No obstante, y por las razones apuntadas al final del resumen, los textos más recomendables sobre lógica aplicada a la informática tratan directamente con la lógica de predicados, y los citaremos en el capítulo 4.

8. Ejercicios

- 8.1. Formalizar las siguientes frases como sentencias proposicionales, y analizar sus tablas de verdad:

"La verdad es una brújula loca que no funciona en este caos de cosas desconocidas" (Baroja).

"Se puede conocer la utilidad de una idea y, sin embargo no acertar a comprender el modo de utilizarla" (Goethe).

"Ese lapso de tiempo, corto si se le mide por el calendario, es interminablemente largo cuando, como yo, se ha galopado a través de él" (Kafka).

"El mismo diablo citará a la Sagrada Escritura si viene bien a sus propósitos" (Shakespeare).

- 8.2. Analizar los siguientes razonamientos:

P1: 'Si no llueve, salgo al campo'.

P2: 'Si salgo al campo, respiro'.

C: 'Respiro sí y sólo si no llueve'.

P1: 'Si un monte se quema, algo suyo se quema'.

P2: 'Algo suyo se quema sí y sólo si es usted descuidado'.

P3: 'Si usted no es descuidado, es acreedor a una felicitación'.

C: 'Si usted no es acreedor a una felicitación, entonces es que un monte se quema'.

P1: 'Si un país es una democracia, el Presidente del Gobierno se elige por sufragio universal'.

P2: 'En España, el Presidente no se elige por sufragio universal'.

C: 'Luego España no es una democracia'.

El mismo anterior, modificando así la primera premisa:

P1: 'Si el Presidente de un país es elegido por sufragio universal, ese país es una democracia'.

8.3. Aplicar la resolución y la refutación a los anteriores razonamientos.

8.4. Aplicar la regla de resolución para inferir cuantas conclusiones sean posibles en los siguientes casos, y comprobar los resultados con lo que se obtiene de las tablas de verdad:

a) *P1*: $p \leftrightarrow q$

P2: $p \rightarrow \neg r$

b) *P1*: $p \leftrightarrow q$

P2: $r \rightarrow \neg p$

(Contrastar estos dos casos con el del ejemplo 5.7.4)

c) *P1*: $p \rightarrow q$

P2: $r \rightarrow p$

P3: $\neg r \rightarrow \neg t$

P4: $\neg(s \wedge \neg r)$

P5: $\neg t \rightarrow s$

d) *P1*: $p \rightarrow q \vee r$

P2: $q \rightarrow p$

Fundamentos de informática



Circuitos lógicos combinacionales

1. Introducción

El hardware puede describirse y estudiarse en varios niveles de abstracción. En el más bajo ("nivel físico-electrónico") se consideran los fenómenos físicos básicos y las propiedades de los materiales (semiconductores, metales, dieléctricos) que explican el funcionamiento de los componentes electrónicos (resistores, diodos, transistores, etc.). En el siguiente ("nivel electrónico-circuital") se hace abstracción de tales fenómenos: se da por supuesta la existencia de los componentes, con comportamiento funcional conocido, y se estudian los circuitos que resultan de su interconexión. Aquí vamos a movernos en el nivel de abstracción inmediatamente superior (el "nivel lógico"): supuesto que disponemos de unos circuitos básicos, las "puertas lógicas", estudiaremos cómo pueden interconectarse para conseguir sistemas con determinado comportamiento. En este nivel, hacemos abstracción tanto de los detalles del nivel "físico-electrónico" como de los del nivel "electrónico-circuital". Por ejemplo, una de las cosas de las

que hacemos abstracción es el valor real de los potenciales eléctricos. En los circuitos digitales se opera siempre con sólo dos valores diferentes de tensión. Según sea la tecnología utilizada, así serán esos dos valores, que, en el nivel lógico, los representaremos por los símbolos "0" y "1". El "0" podría corresponder, en el circuito electrónico, a una tensión de 0 voltios (o, con mayor realismo, a un margen, por ejemplo, de 0 a 1 voltios), y el "1" a 5 voltios (o al margen 3-6). Pero también podría ser al revés (corresponder el "0" a 5v y el "1" a 0v), o podríamos tener otros valores u otros márgenes de tensión totalmente diferentes. Nosotros consideraremos perfectamente definido el comportamiento de los componentes básicos ("puertas") mediante relaciones de entrada-salida que se refieren sólo a los valores lógicos.

Hay dos ideas esenciales que conviene tener muy claras desde el principio para la buena comprensión de este capítulo:

- Cualquier punto de un circuito digital, en un instante determinado, sólo puede estar en uno de dos estados (tensiones o márgenes de tensión) determinados: no hay estados intermedios.
- El valor lógico de un punto de un circuito es la simbolización del estado en que se encuentra ese punto. Este valor lógico puede ser constante (0 o 1: el punto siempre se encuentra en ese estado) o variable (el punto puede estar en un estado u otro, pero en cada instante tiene que estar en alguno de ellos).

En este capítulo estudiaremos los circuitos digitales más sencillos, los llamados "circuitos combinacionales". En ellos, el valor de la salida en cada momento sólo depende de los valores que en ese momento tengan las entradas, y no de los que hubieran podido tener anteriormente. En el Tema "Autómatas" nos ocuparemos de los "circuitos secuenciales", en los que la salida depende de la "historia", es decir, de la sucesión de valores anteriores de las entradas.

Veremos enseguida que el álgebra de Boole, inicialmente propuesta como modelo matemático para la lógica de proposiciones (capítulo 2, apartado 4), es también la herramienta básica para el estudio y el diseño de los circuitos digitales, o circuitos de conmutación, que, por esta misma razón, recordémoslo (capítulo 1, apartado 3), se llaman "circuitos lógicos".

2. Las puertas básicas

Los bloques elementales para la construcción de circuitos lógicos son las "puertas". Este nombre responde al hecho de que pueden tener una o varias entradas pero una sola salida, y esta salida puede tomar el valor lógico "0" ("puerta cerrada") o "1" ("puerta abierta"), dependiendo de los valores lógicos que tengan las entradas. Las tres puertas básicas (más adelante veremos otras) son las llamadas "NOT", "OR" y "AND", y los símbolos más utilizados para repre-

sentarlas en los diagramas son los indicados en la figura 3.1. Su función se corresponde exactamente con la que simbólicamente realizan las conectivas " \neg ", " \vee " y " \wedge " en la lógica de proposiciones. Es decir, si siguiésemos el convenio de representar por " p " y " q " las entradas de una puerta "OR", por ejemplo, la salida estaría representada por la sentencia " $p \vee q$ ", lo que quiere decir que el nivel lógico de esa salida será "0" si los niveles lógicos de las dos entradas son "0", y "1" si una de las entradas, o ambas, tienen el nivel lógico "1".

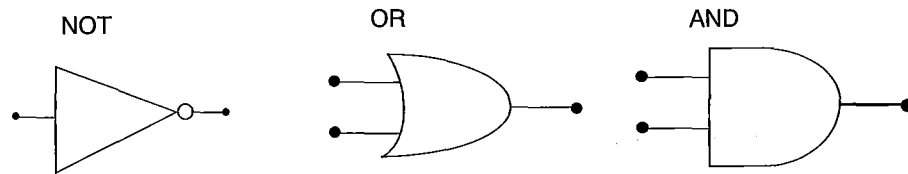


Figura 3.1.

En este capítulo vamos a adoptar, sin embargo, otra notación, más habitual cuando se trabaja con el álgebra de Boole: en lugar de " \neg ", " \vee " y " \wedge " utilizaremos " $-$ ", " $+$ " y " \cdot ", respectivamente. Y para las variables que representan valores lógicos (lo que luego llamaremos "variables booleanas") emplearemos las letras x , y , z , eventualmente con subíndices.

Teniendo en cuenta lo dicho, la figura 3.2, que especifica las funciones de las tres puertas mediante tablas de verdad, se explica por sí sola.

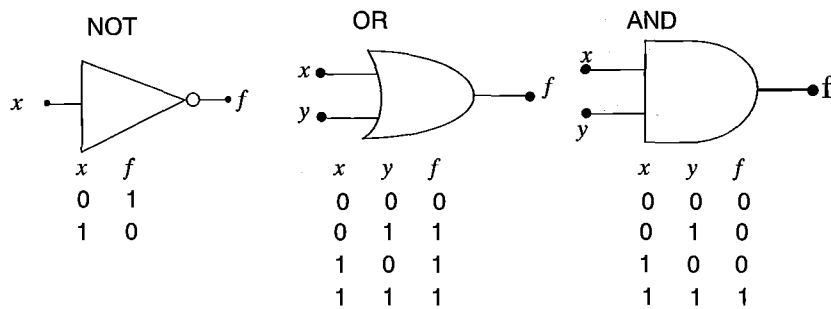


Figura 3.2.

Las puertas "OR" y "AND" pueden tener más de dos entradas. Las operaciones que realizan son, formalmente, las mismas operaciones conocidas del álgebra de Boole, y, por tanto, tienen las mismas propiedades, y, concretamente, la propiedad asociativa. Esto nos permite decir, por ejemplo, que la función de una puerta "AND" de tres entradas (figura 3.3, izquierda) es la misma del circuito formado por dos puertas "AND" de dos entradas conectadas según indica la figura 3.3 (derecha). Y lo mismo podríamos decir para las puertas "OR".

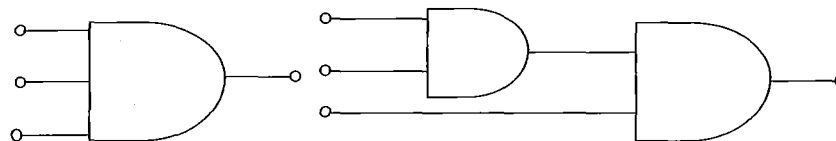


Figura 3.3.

3. Circuitos

Las puertas se pueden interconectar teniendo en cuenta la regla de que la salida de una puerta cualquiera puede servir de entrada a una o varias puertas, pero nunca pueden conectarse juntas dos o más salidas¹. Los sistemas así contruidos serán circuitos lógicos con una o varias entradas y una o varias salidas. Como existe una correspondencia biunívoca entre las operaciones que realizan las puertas y los operadores del álgebra de Boole, si representamos por variables (x_i , y_i , etc.) los valores lógicos de las entradas del circuito, podremos escribir una fórmula para representar cada salida, en la que intervendrán esas variables y los símbolos " \neg ", " $+$ " y " \cdot ". A cada salida de un circuito lógico corresponderá así una fórmula. Por otra parte, el comportamiento del circuito puede especificarse mediante una tabla de verdad para cada salida que nos dé los valores lógicos que toma esa salida para todas y cada una de las posibles combinaciones de valores lógicos de las entradas. La mejor manera de comprender todo esto es a través de ejemplos. En cada uno de los ejemplos que siguen sólo se considera una salida, cuyo valor lógico se simboliza por " f ".

Ejemplo 3.1

Fórmula:

$$f = x + y \cdot z$$

Circuito: El de la figura 3.4..



Figura 3.4.

¹ En realidad, esta regla tiene excepciones: por una parte, el número de entradas que puede alimentar la salida de una puerta tiene una limitación tecnológica; por otra, a veces pueden conectarse directamente las salidas de dos o más puertas, consiguiendo una función "OR" o "AND" "implícita". Pero estas excepciones entran ya en el nivel de descripción "electrónico", más que en el "lógico", por lo que no las consideraremos aquí.

Tabla de verdad:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Ejemplo 3.2

Fórmula:

$$f = x \cdot \bar{y} + x \cdot z + x \cdot y \cdot \bar{z} + \bar{x} \cdot y \cdot z$$

Circuito:

El de la figura 3.5.

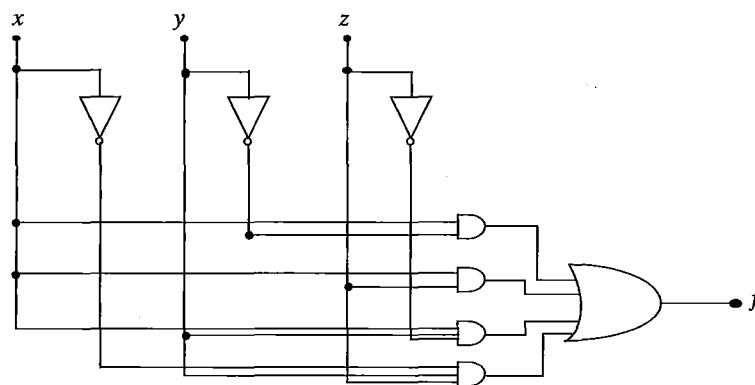


Figura 3.5.

Tabla de verdad:

La misma del ejemplo anterior.

Ejemplo 3.3

Fórmula:

$$f = x_1 \cdot (\overline{x_2} + x_4) + x_3$$

Circuito:

El de la figura 3.6.

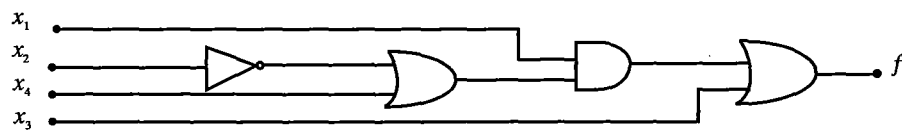


Figura 3.6.

Tabla de verdad:

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Ejemplo 3.4

Fórmula:

$$f = x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot \overline{x_3} \cdot x_4 + x_3$$

Circuito:

El de la figura 3.7.

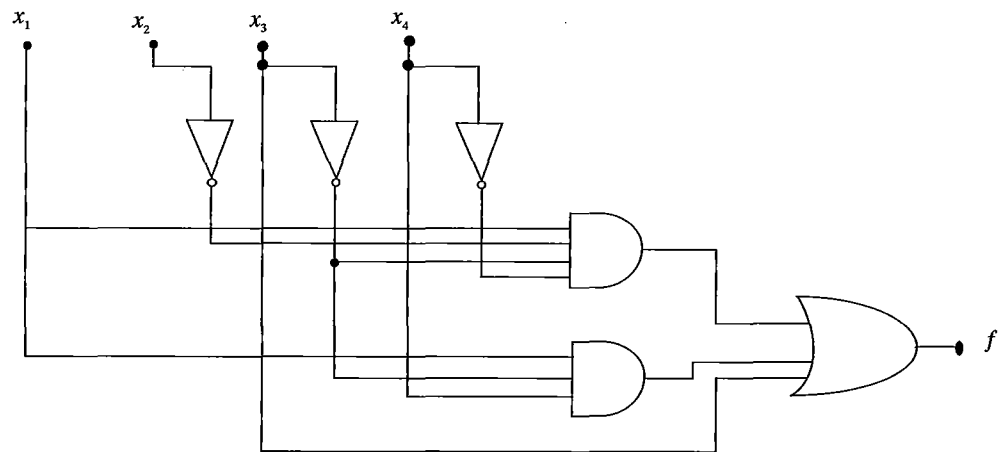


Figura 3.7.

Tabla de verdad:

La misma del ejemplo anterior.

Ejemplo 3.5

Fórmula:

$$f = x \cdot y + (x + y) \cdot z$$

Circuito:

El de la figura 3.8.

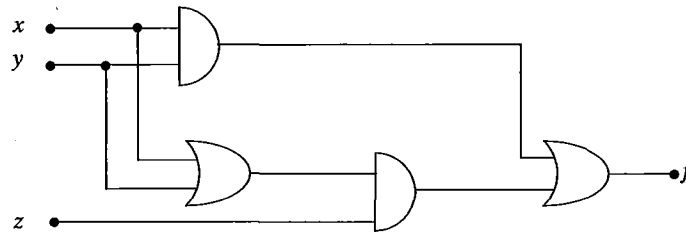


Figura 3.8.

Tabla de verdad:

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Ejemplo 3.6

Fórmula:

$$f = x \cdot y \cdot \bar{z} (x + y) \cdot z$$

Circuito:

El de la figura 3.9.

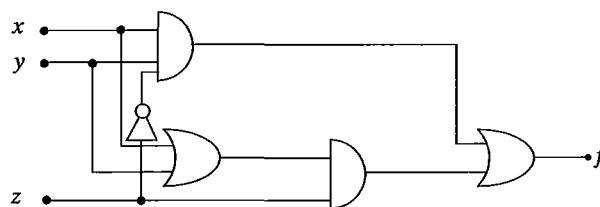


Figura 3.9.

Tabla de verdad:

La misma del ejemplo anterior.

Ejemplo 3.7

Fórmula:

$$f = \bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot z$$

Circuito:

El de la figura 3.10.

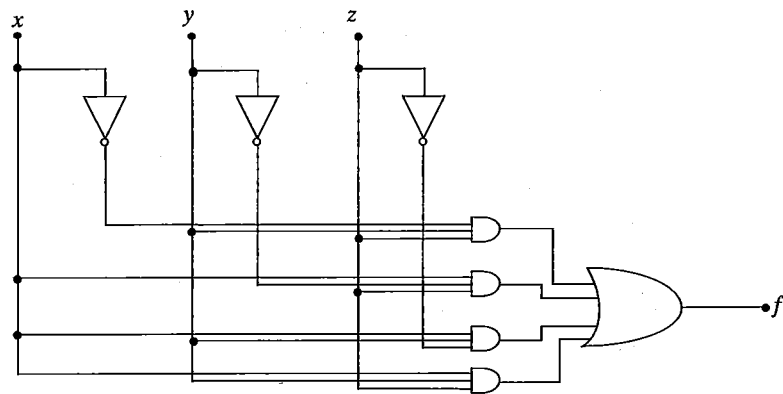


Figura 3.10.

Tabla de verdad:

La misma de los dos ejemplos anteriores.

4. Modelos matemáticos de los circuitos

4.1. Utilidad de los modelos matemáticos

Cuando se aborda la tarea de diseñar un circuito lógico para realizar una determinada función, lo primero es especificar claramente la función. Con frecuencia, esa especificación, que inicialmente puede ser verbal, conduce a una tabla de verdad para cada una de las salidas del circuito. Ahora bien, en los ejemplos anteriores se ve claramente que la misma tabla de verdad puede realizarse mediante circuitos diferentes. Parece obvio que, por muchas razones (coste, tamaño, fiabilidad, etc.), interesará encontrar el circuito que, respetando las

especificaciones, tenga el menor número posible de puertas. Este es el problema de la *minimización*, que trataremos en el siguiente apartado. Para poder abordarlo, necesitamos formalizar algunos conceptos; los modelos matemáticos de los circuitos nos permitirán diseñar éstos de forma óptima.

En los ejemplos hemos comprobado que la función de cada salida de un circuito puede expresarse mediante una tabla de verdad, pero que esta relación no es biunívoca: puede haber varios circuitos con la misma tabla; diremos que todos ellos realizan la misma *función de conmutación*. Por otra parte, también se ve en los mismos ejemplos que a cada circuito podemos asociar de manera biunívoca una "fórmula", construida por variables lógicas y operaciones " \neg ", " $+$ " y " \cdot "; estas fórmulas corresponden a lo que definiremos como *formas booleanas*.

4.2. Funciones de conmutación

Definición 4.2.1. Llamaremos *función de conmutación de orden n* a cualquier aplicación $\{0, 1\}^n \rightarrow \{0, 1\}$, donde $\{0, 1\}^n$ es el producto cartesiano de orden n de $\{0, 1\}$ consigo mismo.

Así, el dominio de una función de conmutación de orden n está formado por todas las n -tuplas que pueden formarse con los elementos 0 y 1, mientras que el rango es solamente $\{0, 1\}$. La forma más natural de representar una determinada función de conmutación es mediante una tabla de verdad. En los ejemplos 3.1 y 3.2 tenemos una función de conmutación de orden 3, en 3.3 y 3.4 una función de conmutación de orden 4, y en los ejemplos 3.5, 3.6 y 3.7 otra función de conmutación de orden 3.

Consideremos el álgebra de Boole binaria $\langle \{0, 1\}, +, \cdot, \neg \rangle$, y llamemos F_n al conjunto de funciones de conmutación de orden n :

$$F_n = \{f: \{0, 1\}^n \rightarrow \{0, 1\}\}$$

Vamos a definir las operaciones "suma", "producto" y "complementación" en el conjunto F_n . Utilizaremos para ellas los mismos símbolos del álgebra de Boole binaria.

Definición 4.2.2. Si f_i y f_j son dos funciones de conmutación de orden n :

$$f_i: X \rightarrow f_i(X); f_j: X \rightarrow f_j(X),$$

$$\text{con } X \in \{0, 1\}^n; f_i(X), f_j(X) \in \{0, 1\},$$

se definen:

- a) la función complementaria de f_i , $\overline{f_i}$ como aquella que hace corresponder a X el elemento complementario del que le corresponde según f_i :

$$\overline{f_i}: X \rightarrow f_i(\bar{X})$$

- b) la función suma de f_i y f_j , $(f_i + f_j)$, como aquella que hace corresponder a X la suma de los elementos que le corresponden según f_i y f_j :

$$(f_i + f_j) : X \rightarrow f_i(X) + f_j(X)$$

- c) la función producto de f_i y f_j , $(f_i \cdot f_j)$, como aquella que hace corresponder a X el producto de los elementos que le corresponden según f_i y f_j :

$$(f_i \cdot f_j) : X \rightarrow f_i(X) \cdot f_j(X)$$

Ejemplo 4.2.3. Llamando f_1 y f_2 a las funciones de conmutación de orden 3 que corresponden a los ejemplos 3.1 (y 3.2) y 3.5 (y 3.6 y 3.7), en la siguiente tabla podemos ver sus correspondientes funciones complementarias, su suma y su producto:

x	y	z	f_1	f_2	\bar{f}_1	\bar{f}_2	$f_1 + f_2$	$f_1 \cdot f_2$
0	0	0	0	0	1	1	0	0
0	0	1	0	0	1	1	0	0
0	1	0	0	0	1	1	0	0
0	1	1	1	1	0	0	1	1
1	0	0	1	0	0	1	1	0
1	0	1	1	1	0	0	1	1
1	1	0	1	1	0	0	1	1
1	1	1	1	1	0	0	1	1

Teorema 4.2.4. $\langle F_n, +, \cdot, \bar{} \rangle$ es un álgebra de Boole cuyos elementos neutros son las funciones que hacen corresponder todas las n -tuplas a 0 o a 1:

$$0 = f_0 \text{ tal que } f_0(X) = 0 \quad \forall X \in \{0, 1\}^n$$

$$1 = f_N \text{ tal que } f_N(X) = 1 \quad \forall X \in \{0, 1\}^n$$

y cuyos elementos atómicos son las 2^n funciones que hacen corresponder todas las n -tuplas a 0, excepto una.

Demostración:

Las operaciones sobre funciones de orden n dan como resultado otras funciones de orden n , por lo que F_n es cerrado bajo las tres operaciones. Por otra

parte, tal como se han definido los elementos neutros, es inmediato que, para cualquier f_i , resulta que $(f_i + f_0) = f_i$ y que $(f_i \cdot f_N) = f_i$. El resto de los axiomas del álgebra de Boole (capítulo 2, apartado 4.1) se cumplen también por el hecho de que las operaciones entre funciones se han definido a partir de las operaciones en el álgebra de Boole $\langle \{0,1\}, +, \cdot, ' \rangle$. Finalmente, es fácil comprobar que las funciones que hacen corresponder todas las n -tuplas, salvo una, a 0, cumplen la definición de elemento atómico (capítulo 2, definición 4.3.1): su producto por cualquier otra función da como resultado o bien f_0 o bien el mismo elemento atómico.

4.3. Formas booleanas

4.3.1. El lenguaje de las formas booleanas

Definición 4.3.1.1. Dada un álgebra de Boole binaria, $\langle \{0,1\}, +, \cdot, ' \rangle$, llamaremos *variables booleanas* a unos símbolos, $x_1, y_1, z_1, x_2, y_2, z_2, \dots$, que representan a los elementos del conjunto $\{0,1\}$. Es decir, una variable booleana puede tomar uno de dos *valores*: 0 o 1. Cuando 0 y 1 representan los valores lógicos de un circuito, las variables booleanas corresponden a las variables lógicas de las que hablábamos en los apartados 1, 2 y 3.

Partiendo del alfabeto formado por el conjunto de variables booleanas y los símbolos "+", "·" y "'" podemos definir un lenguaje siguiendo exactamente los mismos pasos de la lógica de proposiciones (capítulo 2, apartado 2), que ahora nos limitamos a dejar indicados:

- a) Definición de expresión o cadena como secuencia finita de símbolos del alfabeto.
- b) Definición de secuencia de formación.
- c) Definición de sentencia, a la que ahora llamaremos forma booleana.

Las seis "fórmulas" de los ejemplos del apartado 3 son ejemplos de formas booleanas. Como los símbolos "+", "·" y "'" se corresponden, respectivamente, con las puertas "OR", "AND" y "NOT", a cada circuito lógico (o, mejor, a cada salida del mismo) le corresponde una forma booleana, y viceversa.

En realidad, el lenguaje de las formas booleanas es el mismo de la lógica de proposiciones, cambiando las conectivas " \vee ", " \wedge " y " \neg " por los símbolos "+", "·" y "'", respectivamente. El hecho de que las variables y las formas booleanas puedan tomar los valores "0" y "1" de un álgebra de Boole binaria se corresponde con la evaluación binaria de variables proposicionales y de sentencias. Por todo ello, lo que sigue viene a ser una repetición de lo que ya hemos estudiado en el capítulo 2, y lo presentaremos de forma resumida.

4.3.2. Evaluación de formas booleanas

Definición 4.3.2.1. Dado un conjunto de variables booleanas, A , llamaremos *función de asignación de valores* (o, simplemente, *asignación*) a una aplicación del conjunto A en el conjunto $\{0,1\}$:

$$v: A \rightarrow \{0, 1\}$$

El concepto es exactamente el mismo de "evaluación binaria" de la lógica de proposiciones (como vimos en el capítulo 2, apartado 3). Aquí lo hemos restringido de entrada al caso binario porque lo aplicaremos exclusivamente a circuitos con dos estados, pero podríamos haberlo planteado de un modo más general, de modo que el modelo matemático pudiera servir para circuitos con tres o más estados.

Definición 4.3.2.2. Dados un conjunto de formas booleanas construidas a partir de un conjunto de variables A y una asignación v_i , llamaremos *función de evaluación*, V_i , (o, simplemente, *evaluación*) a la extensión del dominio de la asignación al conjunto de formas booleanas. Este otro concepto es exactamente el mismo de "evaluación de sentencias" (capítulo 2, teorema 3.1.4), y, lo mismo que allí, se puede demostrar que esta extensión es única. Las tablas de verdad de los ejemplos del apartado 3 representan todas las funciones de evaluación posibles de las correspondientes formas booleanas.

4.3.3. Equivalencia de formas booleanas

Definición 4.3.3.1. Llamemos B_n al conjunto (infinito) de formas booleanas construidas con n variables booleanas diferentes, y sean $A, B \in B_n$. Diremos que A y B son equivalentes, $A \equiv B$, si y sólo si sus evaluaciones son iguales para todas las asignaciones:

$$(\forall v_i) (V_i(A) = V_i(B))$$

El concepto es exactamente el mismo de la "equivalencia entre sentencias" (capítulo 2, apartado 3.5). Llamaremos ahora $C_n = B_n / \equiv$ al conjunto de clases de equivalencia que resultan de la partición de B_n por la relación de equivalencia. El número de clases de equivalencia es finito: $\text{card}(C_n) = 2^{2^n}$, pero dentro de cada una hay infinitas formas booleanas diferentes (en efecto, x es equivalente a $x + x$, y a $x \cdot x$, y a $x + x + x$, ...).

Definición 4.3.3.2. Las evaluaciones de una clase de equivalencia, $V_k(c_i)$ ($c_i \in C_n$), son las evaluaciones de una cualquiera de las formas booleanas que pertenecen a esa clase:

$$V_k(c_i) = V_k(A) \quad (A \in c_i)$$

La forma booleana del ejemplo 3.1 es equivalente a la del ejemplo 3.2, la del 3.3 es equivalente a la del 3.4, y las tres formas booleanas de los ejemplos 3.5, 3.6 y 3.7 son equivalentes entre sí.

4.3.4. Algebra de Boole de las clases de equivalencia entre formas booleanas

Definamos primero las operaciones "suma", "producto" y "complementación" en el conjunto C_n . Utilizaremos los mismos símbolos del álgebra de Boole binaria.

Definición 4.3.4.1. Si $c_i, c_j \in C_n$ son dos clases de equivalencia de formas booleanas con n variables, se definen las clases complementaria, suma y producto del siguiente modo:

a) \bar{c}_i es aquella clase tal que

$$(\forall v_k) [V_k(\bar{c}_i) = \overline{V_k(c_i)}]$$

b) $c_i + c_j$ es aquella clase tal que

$$(\forall v_k) [V_k(c_i + c_j) = V_k(c_i) + V_k(c_j)]$$

c) $c_i \cdot c_j$ es aquella clase tal que

$$(\forall v_k) [V_k(c_i \cdot c_j) = V_k(c_i) \cdot V_k(c_j)]$$

Teorema 4.3.4.2. $\langle C_n, +, \cdot, \bar{} \rangle$ es un álgebra de Boole cuyos elementos neutros son c_0 (clase de equivalencia cuyas evaluaciones son 0 para todas las asignaciones) y c_N (clase de equivalencia cuyas evaluaciones son 1 para todas las asignaciones) y cuyos elementos atómicos son las 2^n clases de equivalencia tales que sus evaluaciones son 0 para todas las asignaciones salvo una.

Este teorema es, en realidad, el mismo que habíamos visto para las clases de equivalencia entre sentencias (capítulo 2, apartado 4.2), y su demostración es la misma que allí apuntábamos (como también para el teorema 4.2.4 de este capítulo): basta con ver que se satisfacen los axiomas del álgebra de Boole. En cuanto a los elementos atómicos, basta con ver que se cumplen las condiciones de su definición (definición 4.3.1 del capítulo 2): el producto de cualquier c_i por un elemento atómico da como resultado c_0 o c_i .

4.3.5. Formas canónicas

Consideremos el álgebra de Boole $\langle C_n, +, \cdot, \bar{} \rangle$ de las formas booleanas generadas por las variables x_1, x_2, \dots, x_n . Denominaremos l_i a una "metavariante", que puede valer x_i o \bar{x}_i (es lo que en el capítulo 2 llamábamos un "literal").

Llamaremos *producto canónico* a toda forma booleana compuesta por el producto de todas las variables complementadas o no:

$$P_i = l_1 \cdot l_2 \cdot \dots \cdot l_n = \prod_{j=1}^n l_j$$

(donde " Π " representa el producto booleano).

Un cálculo combinatorio elemental nos da que el número de productos canónicos diferentes con n variables booleanas es 2^n .

Definición 4.3.5.2. Llamaremos *forma canónica* a toda forma booleana compuesta por una suma de productos canónicos diferentes entre sí.

Teorema 4.3.5.3. Toda clase de equivalencia en C_n puede representarse mediante su forma canónica, que es única para esa clase de equivalencia.

Demostración:

- a) Como en un producto canónico intervienen todas las variables, sus evaluaciones serán siempre "0" excepto para una determinada asignación: aquella que asigne el valor "1" a las variables sin complementar y el valor "0" a las variables complementadas. Por tanto, cada uno de los 2^n productos canónicos sirve para representar a uno de los 2^n elementos atómicos de $\langle C_n, +, \cdot, \bar{}, } \rangle$ (teorema 4.3.4.2).
- b) Según el teorema 4.3.2 enunciado en el capítulo 2, todo elemento perteneciente a un álgebra de Boole cuyo valor sea distinto de 0 puede expresarse de manera única como suma de elementos atómicos. Así pues, cualquier clase de equivalencia distinta de c_0 puede expresarse de manera única como suma de elementos atómicos, y, por consiguiente, podrá representarse de manera única como suma de productos canónicos, es decir, en la primera forma canónica.

Existe una notación abreviada para escribir formas canónicas, que se basa en asociar un número decimal a cada uno de los productos canónicos. Este número es el que resulta de considerar como un número binario la combinación de "ceros" y "unos" de las variables para la cual la evaluación resultante del producto es "1". Así por ejemplo, $\bar{x} \cdot y \cdot z$ es "1" para $x = 0, y = 1, z = 1$, y "011" en binario es "3" en decimal. Del mismo modo, $x \cdot \bar{y} \cdot z$ es "1" para $x = 1, y = 0, z = 1$, y "101" en binario es "5" en decimal. La notación abreviada de la forma canónica $\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z$ sería: $\Sigma(3,5)$ (suma de los productos canónicos de los números "3" y "5").

Ejemplo 4.3.5.4. Si consideramos las formas booleanas que pueden formarse con dos variables, x e y , el número de clases de equivalencia es dieciséis (lo

mismo que entre las sentencias construidas con dos variables proposicionales y con evaluaciones binarias, véase el apartado 3.5 del capítulo 2). Los productos canónicos son cuatro: $x \cdot y$, $x \cdot \bar{y}$, $\bar{x} \cdot y$, $\bar{x} \cdot \bar{y}$ (en notación abreviada, "3", "2", "1" y "0", respectivamente), que corresponden a los cuatro elementos atómicos c_1 (clase de equivalencia de las formas cuyas evaluaciones son "0" excepto para $x = 1, y = 1$), c_2 (idem, excepto para $x = 1, y = 0$), c_4 (idem, excepto para $x = 0, y = 1$) y c_8 (idem, excepto para $x = 0, y = 0$). Cualquier otra clase de equivalencia puede representarse como suma de dos o más productos canónicos. Por ejemplo, la c_9 tiene evaluación "1" para dos asignaciones: para $x = 0, y = 0$, y para $x = 1, y = 1$; su forma canónica será la suma de los dos productos canónicos correspondientes: $\bar{x} \cdot \bar{y} + x \cdot y$ (o, en notación abreviada, $\Sigma(0,3)$).

Ejemplo 4.3.5.5. Con tres variables booleanas tendremos $2^3 = 8$ elementos atómicos (representados por los productos canónicos: $(x \cdot y \cdot z, x \cdot y \cdot \bar{z}, \dots, \bar{x} \cdot \bar{y} \cdot \bar{z})$) y $2^8 = 256$ clases de equivalencia. Las dos formas booleanas de los ejemplos 3.1 y 3.2 pertenecen a la misma clase. Para ver cuál es su forma canónica basta con mirar en la tabla de verdad, donde están indicadas todas las evaluaciones posibles, qué productos canónicos son los responsables de los cinco evaluaciones "1" que aparecen en dicha tabla, y sumarlos. El resultado es el siguiente:

$$\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot z$$

(en notación abreviada: $\Sigma(3,4,5,6,7)$). Del mismo modo, el lector puede comprobar que la forma canónica que corresponde a los ejemplos 3.5, 3.6 y 3.7 es la forma booleana del último de ellos (en notación abreviada: $\Sigma(3,5,6,7)$).

Ejemplo 4.3.5.6. Con cuatro variables booleanas habrá $2^4 = 16$ elementos atómicos (productos canónicos $x_1 \cdot x_2 \cdot x_3 \cdot x_4, x_1 \cdot x_2 \cdot x_3 \cdot \bar{x}_4, \dots, \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4$) y $2^{16} = 65536$ clases de equivalencia. El procedimiento para escribir la primera forma canónica de una clase cualquiera es el mismo de antes: sumar los productos canónicos que corresponden a las evaluaciones "1". A la vista de la tabla de verdad de los ejemplos 3.3 y 3.4, obtenemos la correspondiente forma canónica, que, escrita en notación abreviada, es $\Sigma(2,3,6,7,8,9,10,11,13,14,15)$. Obsérvese que el paso de la tabla de verdad a la notación abreviada es inmediato: basta con numerar sus filas de 0 a $2^n - 1$, e incluir todos los números de las filas que correspondan a una evaluación de "1".

4.4. Relación entre los dos modelos matemáticos

En los últimos ejemplos hemos utilizado las tablas de verdad como representaciones de las evaluaciones de una forma booleana (y de todas las que están

en su misma clase de equivalencia). Por otra parte, en el apartado 4.2 hacíamos uso de las mismas tablas de verdad como ejemplos de funciones de conmutación. Esta dualidad tiene un fundamento matemático. En efecto, el álgebra de Boole de las funciones de conmutación de orden n , $\langle F_n, +, \cdot, \bar{} \rangle$, tiene 2^{2^n} elementos, ya que el número de n -tuplas diferentes en $\{0, 1\}^n$ es $k = 2^n$, y hay 2^k combinaciones distintas para aplicar cada una de esas k n -tuplas en $\{0, 1\}$. Por otra parte, el número de clases de equivalencia en el álgebra de Boole de las formas booleanas con n variables, $\langle C_n, +, \cdot, \bar{} \rangle$ es también 2^{2^n} : $k = 2^n$ productos canónicos con los que pueden escribirse 2^k primeras formas canónicas diferentes. Por tanto, y de acuerdo con el teorema 4.3.4 enunciado en el capítulo 2, ambas álgebras de Boole son isomorfas, es decir, existe una correspondencia biunívoca entre cada función de conmutación de orden n y cada clase de equivalencia de formas booleanas de n variables, y esta correspondencia es tal (definición 4.3.3 del capítulo 2) que si f_i y f_j están en correspondencia con c_i y c_j , respectivamente, entonces f_i está en correspondencia con c_i y $(f_i + f_j)$ y $f_i \cdot f_j$ están en correspondencia con $(c_i + c_j)$ y $(c_i \cdot c_j)$, respectivamente.

La importancia de esta base teórica para la aplicación práctica al diseño de circuitos lógicos es la que insinuábamos en el apartado 4.1. Normalmente, tendremos la especificación de cada salida del circuito como una función de conmutación (o, equivalentemente, como una primera forma canónica). Sabemos que a esa función de conmutación le corresponde toda una clase de equivalencia de formas booleanas con n variables, dentro de la cual hay infinitas formas booleanas diferentes; todas ellas corresponden a la misma función de conmutación, pero a cada una le corresponde un circuito lógico diferente. Todos estos circuitos lógicos realizarán la misma función, y lo que nos interesa es elegir el más sencillo. Entramos así en el problema de la minimización.

5. Minimización

5.1. Principios

Para encontrar formas booleanas equivalentes a una dada y más sencillas que ella podemos utilizar dos equivalencias:

a) Reducción de términos adyacentes:

Si A es una forma booleana cualquiera y x una variable booleana, $A \cdot x + A \cdot \bar{x} \equiv A$. Las formas booleanas que sólo difieren en una variable, que aparece complementada en una y sin complementar en la otra, se llaman "términos adyacentes".

b) Idempotencia:

Si A es una forma booleana cualquiera, $A + A \equiv A$.

Por ejemplo, consideremos la forma booleana de tres variables:

$$\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot z$$

Observamos que el último producto es adyacente a cada uno de los otros tres (pero éstos no lo son entre sí). Podemos reducirlo con uno cualquiera de ellos; si lo hacemos con el primero, resulta:

$$y \cdot z + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z}$$

con el segundo:

$$\bar{x} \cdot y \cdot z + x \cdot z + x \cdot y \cdot \bar{z}$$

y con el tercero:

$$\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z + x \cdot y$$

Estas tres formas booleanas son equivalentes entre sí, y equivalentes a la primera. Pero se puede conseguir otra con menos operaciones si previamente "complicamos" la forma de partida teniendo en cuenta la propiedad de idempotencia: se puede sumar dos veces el producto $x \cdot y \cdot z$, y cada uno de estos tres productos se podrá reducir con uno de los tres primeros. El resultado es:

$$x \cdot y + x \cdot z + y \cdot z$$

que es la "forma mínima en suma de productos". Su realización como circuito requiere una puerta OR de tres entradas y tres puertas AND de dos entradas. Una pequeña reducción se consigue teniendo en cuenta la propiedad distributiva, que nos permite escribir esta otra forma booleana equivalente:

$$x \cdot (y + z) + y \cdot z$$

5.2. Método de Karnaugh

Es evidente que en casos más complicados que el del ejemplo que acabamos de considerar no resulta sencillo comprobar qué elementos conviene "desdoblar" para poder aplicar luego reducciones. Se han propuesto diversos métodos para minimizar de modo sistemático. Algunos son numéricos, y conducen a un algoritmo que puede programarse. Aquí veremos un método gráfico que es el más sencillo y también el más conocido (aunque prácticamente deja de tener utilidad para formas booleanas con más de seis variables): el método de las *tablas de Karnaugh*.

Una tabla de Karnaugh no es otra cosa que una presentación alternativa de la misma información contenida en una tabla de verdad. La tabla de Karnaugh es de doble entrada, y tiene las asignaciones colocadas de tal modo que las que

corresponden a productos canónicos adyacentes están físicamente contiguas. En la figura 3.11 pueden verse las disposiciones de las tablas de Karnaugh para los casos de tres, cuatro y cinco variables booleanas. Cada casilla corresponde a una línea de la tabla de verdad, y se pondrá en ella un "0" o un "1". En la figura 3.11 hemos numerado las casillas con los números de los productos canónicos que corresponden a un "1" en esa casilla, lo cual es muy útil cuando se representa una forma canónica expresada en notación abreviada.

$\begin{matrix} xy \\ z \end{matrix}$		00	01	11	10
		0	2	6	4
0		0	2	6	4
1		1	3	7	5

$\begin{matrix} x_1x_2 \\ x_3x_4 \end{matrix}$		00	01	11	10
		00	4	12	8
00		0	4	12	8
01		1	5	13	9
11		3	7	15	11
10		2	6	14	10

$\begin{matrix} x_2x_3 \\ x_4x_5 \end{matrix}$		00	01	11	10
		00	4	12	8
00		0	4	12	8
01		1	5	13	9
11		3	7	15	11
10		2	6	14	10

$\begin{matrix} x_2x_3 \\ x_4x_5 \end{matrix}$		00	01	11	10
		16	20	28	24
00		16	20	28	24
01		17	21	29	25
11		19	23	31	27
10		18	22	30	26

Figura 3.11.

Veamos, mediante algunos ejemplos, cómo se procede para simplificar utilizando las tablas:

Ejemplo 5.2.1. La tabla de Karnaugh para el ejemplo considerado más arriba:

$$\bar{x} \cdot y \cdot z + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} + x \cdot y \cdot z = \Sigma(3,5,6,7)$$

es la que puede verse en la figura 3.12. Como se indica en ella, se agrupa uno de los "1" (el que corresponde a $x \cdot y \cdot z$) con los tres que le son adyacentes.

Cuando se agrupan dos "1" desaparece una variable: la que tiene asignación "0" en un caso y "1" en otro. El resultado es: $x \cdot y + x \cdot z + y \cdot z$.

Ejemplo 5.2.2. Si consideramos la forma canónica $\Sigma(3,4,5,6,7)$ tenemos la tabla de la figura 3.13. Vemos en ella que hay dos productos, $x \cdot y, x \cdot \bar{y}$ resultado de reducir dos parejas de productos adyacentes, y que, a su vez, son adyacentes y se reducen a x . A efectos prácticos, cuando vemos cuatro "1"

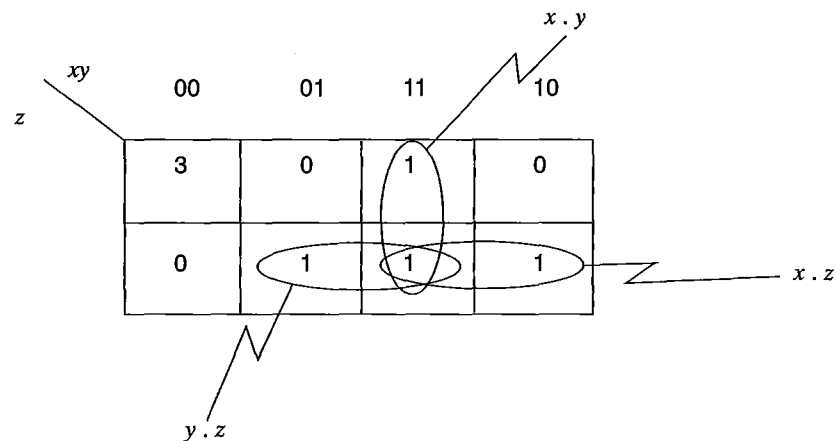


Figura 3.12.

formando un cuadrado eliminamos dos variables: las que tienen valor "0" en unos casos y "1" en otro. El resultado final para este ejemplo es: $x + y \cdot z$.

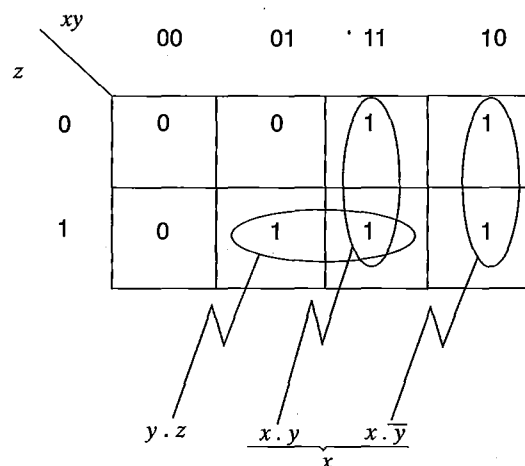


Figura 3.13.

Ejemplo 5.2.3. La tabla de la figura 3.14 corresponde a la forma canónica $\Sigma(1,3,5,6,7)$. Vemos que ocurre algo parecido a lo anterior: también pueden reducirse cuatro "1" que estén en la misma línea. Resultado: $z + x \cdot y$.

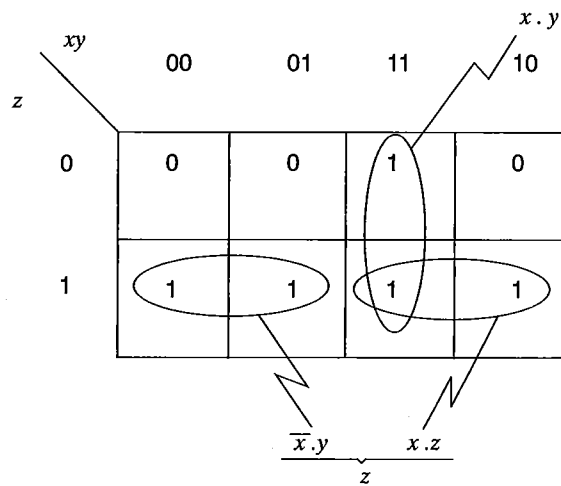


Figura 3.14.

Ejemplo 5.2.4. En la tabla de la figura 3.15, correspondiente a la forma canónica $\Sigma(0,1,4,5,7)$ tenemos otro hecho importante: las casillas de los bordes son también "adyacentes" a las del borde contrario. Resultado de este ejemplo: $y + x \cdot z$.

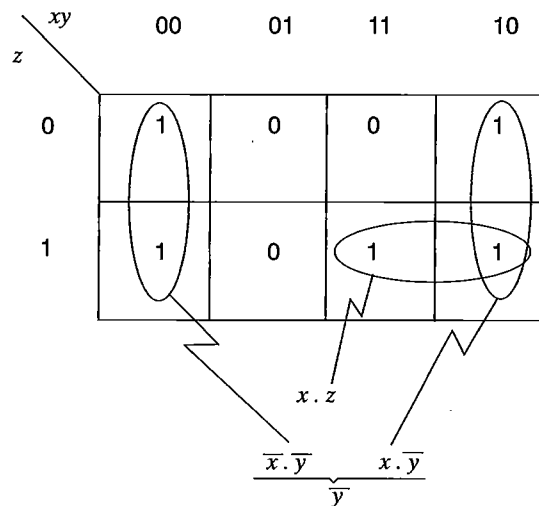


Figura 3.15.

Ejemplo 5.2.5. En la figura 3.16 podemos ver un ejemplo de minimización de una forma booleana de cuatro variables: $\Sigma(0,8,9,10,11,14)$.

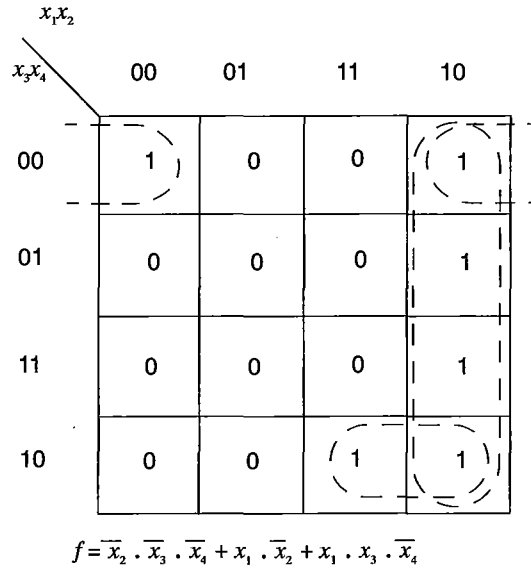


Figura 3.16.

Ejemplo 5.2.6. Si recordamos los ejemplos 3.3 y 3.4 y construimos su correspondiente tabla de Karnaugh (figura 3.17) vemos que la forma mínima (en producto de sumas) es $x_1 \cdot x_2 + x_1 \cdot x_4 + x_3$.

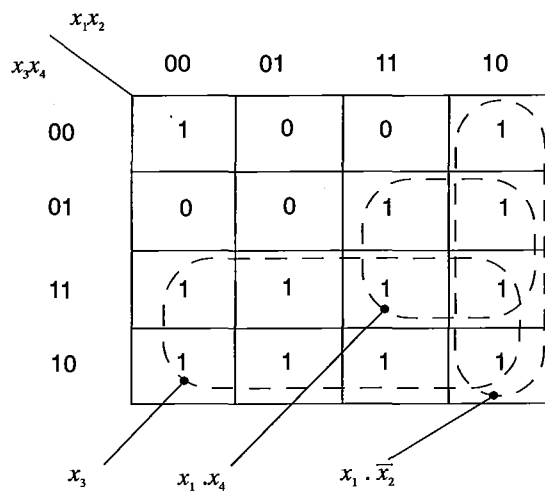


Figura 3.17.

Ejemplo 5.2.7. En la figura 3.18 podemos ver otras posibles agrupaciones en tablas de Karnaugh de cuatro variables.

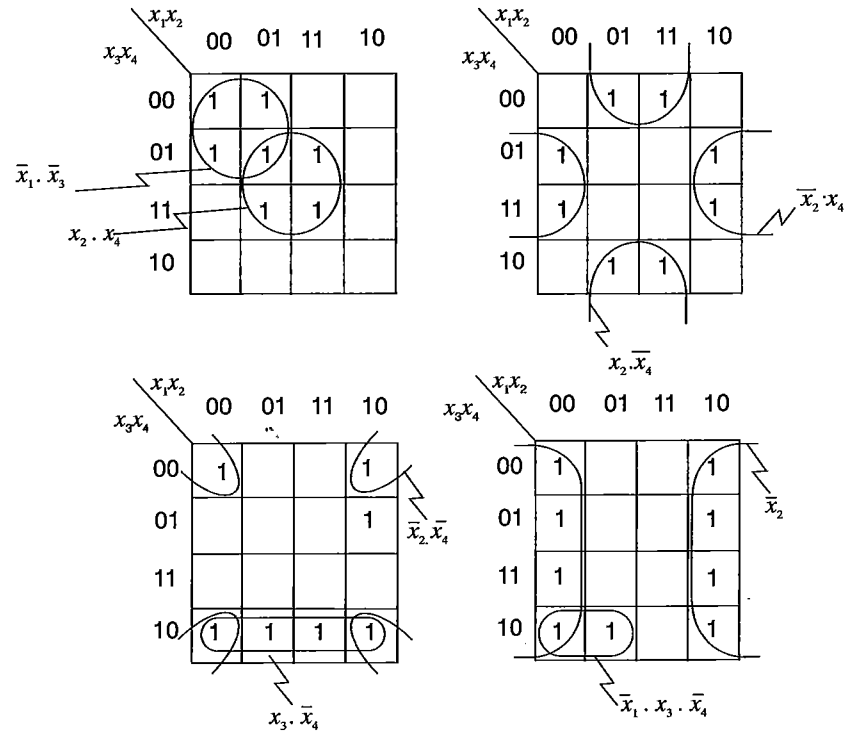


Figura 3.18.

Ejemplo 5.2.8. La figura 3.19 es un ejemplo de cómo se utiliza la tabla de Karnaugh de cinco variables. Basta considerar que las casillas (o los grupos) que ocupan la misma posición en las dos tablas son adyacentes.

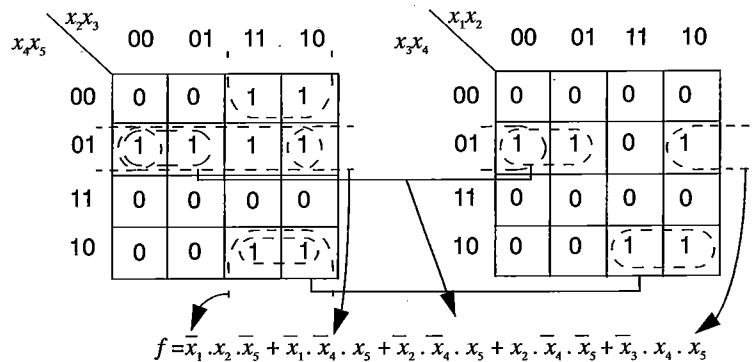


Figura 3.19.

En resumen, el método de Karnaugh consiste en *recubrir* todos los "1" que aparecen en la tabla con el menor número posible de grupos (menor número total de sumandos) y de modo que cada grupo sea lo más grande posible (menor número de variables en cada uno de los productos que forman los sumandos).

6. Ejemplos de aplicación

6.1. Máquina de escrutinio

Supongamos que hay un comité formado por cuatro miembros, de los que uno es presidente, y que las decisiones se toman por mayoría simple, decidiendo el voto del presidente cuando existe empate. Se trata de diseñar una máquina con cuatro entradas (un pulsador para cada miembro) cuya salida dé el resultado de la votación. Llamando A , B , C , y D a las variables lógicas que representan el estado del pulsador de cada miembro (donde A es el que corresponde al presidente) y S a la que representa la salida del circuito, la especificación viene dada por la siguiente tabla de verdad:

A	B	C	D	S
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

En la figura 3.20(a) tenemos esta misma información representada en una tabla de Karnaugh. Vemos que pueden formarse tres grupos de cuatro "1" y un grupo de dos, con el resultado:

$$S = A \cdot B + A \cdot C + A \cdot D + B \cdot C \cdot D = A \cdot (B + C + D) + B \cdot C \cdot D$$

El circuito correspondiente es el dibujado en la figura 3.20(b), si bien para este

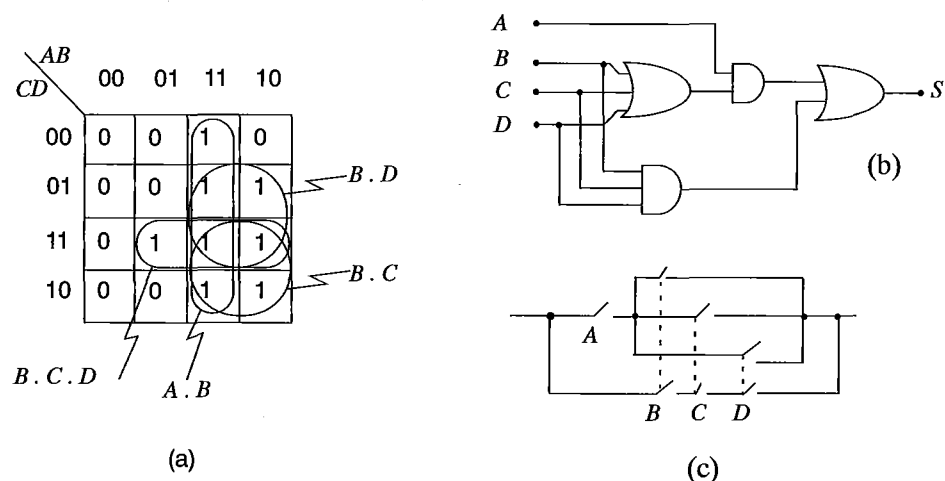


Figura 3.20.

caso resultaría más económico basarse en un simple circuito con interruptores como muestra la figura 3.20(c). (Obsérvese que la conexión en serie de interruptores realiza una función de conmutación correspondiente a un producto lógico, y la combinación en paralelo realiza la de una suma.)

6.2. Alarma para incendios

Tenemos un detector de llamas, un detector de humos y dos detectores de temperatura distribuidos por una sala. Las salidas de esos detectores las simbolizamos, respectivamente, por las variables lógicas A , B , C y D , con valor "0" en caso de normalidad y "1" en caso de detección positiva. Suponemos que el detector de llamas (A) no da "falsos positivos" pero sí "falsos negativos" (es decir, si $A = 1$ la alarma debe dispararse, pero es posible que la alarma también deba dispararse con $A = 0$). Los otros tres detectores pueden fallar tanto en el caso positivo (salida "1" sin incendio) como en el negativo (salida "0" con incendio); consideramos que para que se confirme la alarma es necesario y suficiente que den detección positiva el de humos (B) y uno al menos de los de temperatura (C

o D). De acuerdo con todo esto, podemos especificar el circuito mediante esta tabla:

A	B	C	D	S
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

En la figura 3.21(a) podemos ver los agrupamientos, que conducen a la forma booleana minimizada:

$$S = A + B \cdot C + B \cdot D = A + B \cdot (C + D)$$

y al circuito de la figura 3.21(b)

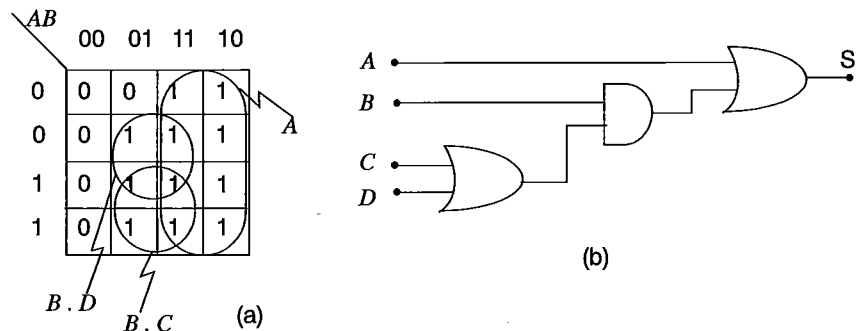


Figura 3.21.

6.3. Etapa de sumador binario

Este ejemplo nos va a permitir ilustrar un caso interesante en la minimización de formas booleanas: el de aquellas que corresponden a *funciones de conmutación incompletamente especificadas*. Este caso aparece cuando se da la circunstancia de que hay combinaciones de valores de las variables de entrada que o bien sabemos que nunca se van a producir o bien, cuando se producen, nos es indiferente el valor que pueda tomar la salida.

Una etapa de sumador binario es un circuito que suma dos dígitos binarios teniendo en cuenta el posible acarreo o arrastre de la suma de los dos dígitos de peso inmediatamente inferior (que pueden haberse sumado en otra etapa). Por tanto, tendrá tres entradas binarias: x , y , r' , correspondientes, respectivamente, a los dos dígitos a sumar y al arrastre de la etapa anterior, y dos salidas: s (suma) y r (arrastre producido). Un sumador binario paralelo de n bits constará de n etapas (de las que la primera no necesita la entrada r'), en las que la salida r de cada una se conecta a la entrada r' de la siguiente.

Si analizamos las posibilidades de x , y , r' , y para cada una anotamos los valores que deben tomar s y r , llegamos a las siguientes tablas de verdad:

x	y	r'	s	r
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

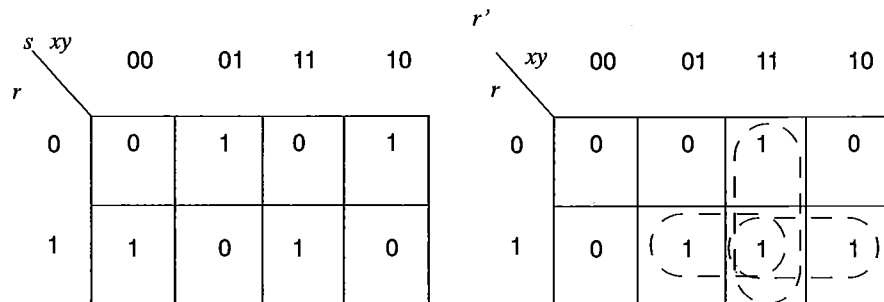


Figura 3.22.

De las tablas de Karnaugh de la figura 3.22 obtenemos las formas booleanas:

$$s = \bar{x} \cdot \bar{y} \cdot r' + \bar{x} \cdot y \cdot \bar{r}' + x \cdot y \cdot r' + x \cdot \bar{y} \cdot \bar{r}'$$

$$r = x \cdot y + r' \cdot (x + y)$$

(La forma correspondiente a r es la que ya habíamos considerado como ejemplo en el apartado 5.1; $x \cdot y$ es el "arrastre generado" en esta etapa y $r' \cdot (x + y)$ es el "arrastre propagado" desde la anterior a la siguiente).

Se observará que s no se ha podido simplificar, y se ha expresado por su forma canónica. Sin embargo, podemos reducir el número de puertas del circuito global si tenemos en cuenta que, en realidad, estamos diseñando dos circuitos: uno para s y otro para r ; cabe entonces pensar en utilizar r como entrada adicional para s , que será así una función de cuatro entradas (x, y, r', r), como muestra la figura 3.23(a). Si, siguiendo esa idea, tratamos a s como una función de cuatro variables,

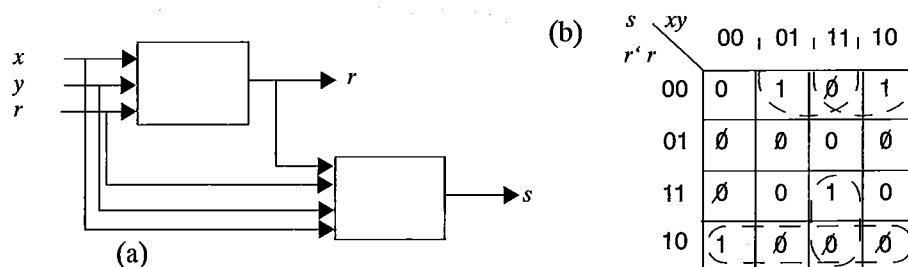


Figura 3.23.

x, y, r', r , lo primero que encontramos es que existen cuádruplas para las que s no está definida, por ejemplo, para $x = 0, y = 0, r' = 0, r = 1$; y s no está definida en este caso (como en otros similares) porque esa combinación es imposible, ya que si x, y, r' valen las tres "0", entonces necesariamente $r = 0$. Para estas combinaciones de entrada, *que nunca se van a presentar* a la entrada del circuito de s ,

podemos tomar para s el valor "0" o "1" indiferentemente, y esto lo representaremos en la tabla de Karnaugh con el símbolo " \emptyset ". Agruparemos los " \emptyset " con los "1" o no, según convenga mejor a efectos de formar el menor número de grupos lo más grandes posibles. Con los grupos que pueden verse en la figura 3.23(b) llegamos a la forma booleana:

$$s = r' \cdot \bar{r} + y \cdot \bar{r} + x \cdot \bar{r} + x \cdot y \cdot r' = \bar{r} \cdot (x + y + r') + x \cdot y \cdot r'$$

El circuito global será el dibujado en la figura 3.24..

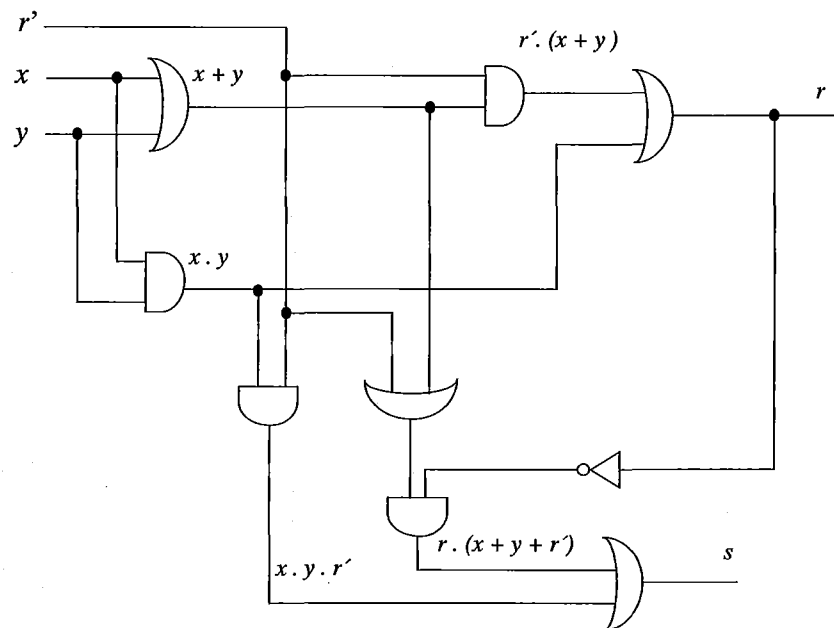


Figura 3.24.

6.4. Multiplicador binario de dos bits

Se trata ahora de diseñar un circuito para multiplicar dos números enteros de valor comprendido entre 0 y 3. Tendrá, por tanto, cuatro entradas (dos para los dos bits que representan a cada uno de los multiplicandos) y cuatro salidas (puesto que el máximo resultado es $3 \cdot 3 = 9$, que en binario puro es 1001). Llamemos x_0 , x_1 a las entradas correspondientes a los bits de peso 0 y 1 de uno de los multiplicandos, y_0 , y_1 a los del otro, y z_0 a z_3 a las salidas correspondientes a los bits de peso 0 a 3 del resultado. Analizando todas las posibilidades, llegamos a esta tabla de verdad (a la izquierda de cada fila figura la correspondiente operación en decimal):

$x * y = z$	x_1	x_0	y_1	y_0	z_3	z_2	z_1	z_0
$0 * 0 = 0$	0	0	0	0	0	0	0	0
$0 * 1 = 0$	0	0	0	1	0	0	0	0
$0 * 2 = 0$	0	0	1	0	0	0	0	0
$0 * 3 = 0$	0	0	1	1	0	0	0	0
$1 * 0 = 0$	0	1	0	0	0	0	0	0
$1 * 1 = 1$	0	1	0	1	0	0	0	1
$1 * 2 = 2$	0	1	1	0	0	0	1	0
$1 * 3 = 3$	0	1	1	1	0	0	1	1
$2 * 0 = 0$	1	0	0	0	0	0	0	0
$2 * 1 = 2$	1	0	0	1	0	0	1	0
$2 * 2 = 4$	1	0	1	0	0	1	0	0
$2 * 3 = 6$	1	0	1	1	0	1	1	0
$3 * 0 = 0$	1	1	0	0	0	0	0	0
$3 * 1 = 3$	1	1	0	1	0	0	1	1
$3 * 2 = 6$	1	1	1	0	0	1	1	0
$3 * 3 = 9$	1	1	1	1	1	0	0	1

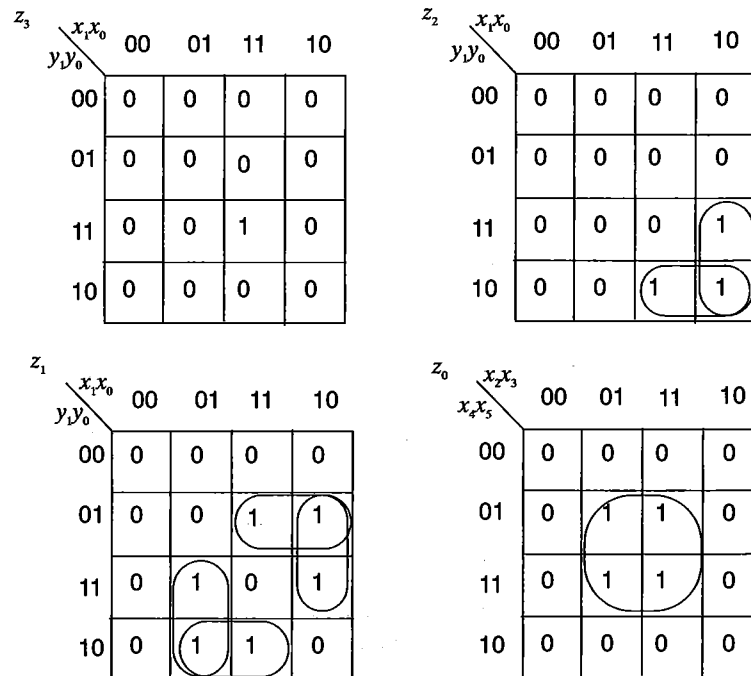


Figura 3.25.

De las tablas de Karnaugh (figura 3.25) obtenemos las formas booleanas:

$$z_3 = x_1 \cdot x_0 \cdot y_1 \cdot y_0$$

$$z_2 = x_1 \cdot \bar{x}_0 \cdot y_1 + x_1 \cdot y_1 \cdot \bar{y}_0 = x_1 \cdot y_1 \cdot (\bar{x}_0 + \bar{y}_0)$$

$$z_1 = \bar{x}_1 \cdot x_0 \cdot y_1 + x_0 \cdot y_1 \cdot \bar{y}_0 + x_1 \cdot \bar{x}_0 \cdot y_0 + x_1 \cdot \bar{y}_1 \cdot y_0 =$$

$$= x_0 \cdot y_1 \cdot (\bar{x}_1 + \bar{y}_0) + x_1 \cdot y_0 \cdot (\bar{x}_0 + \bar{y}_1)$$

$$z_0 = x_0 \cdot y_0$$

y, de ellas, el circuito de la figura 3.26. Como ejercicio, sugerimos al lector que trate de encontrar otras formas booleanas más sencillas para z_1 y z_2 considerando como entradas adicionales las otras dos salidas.

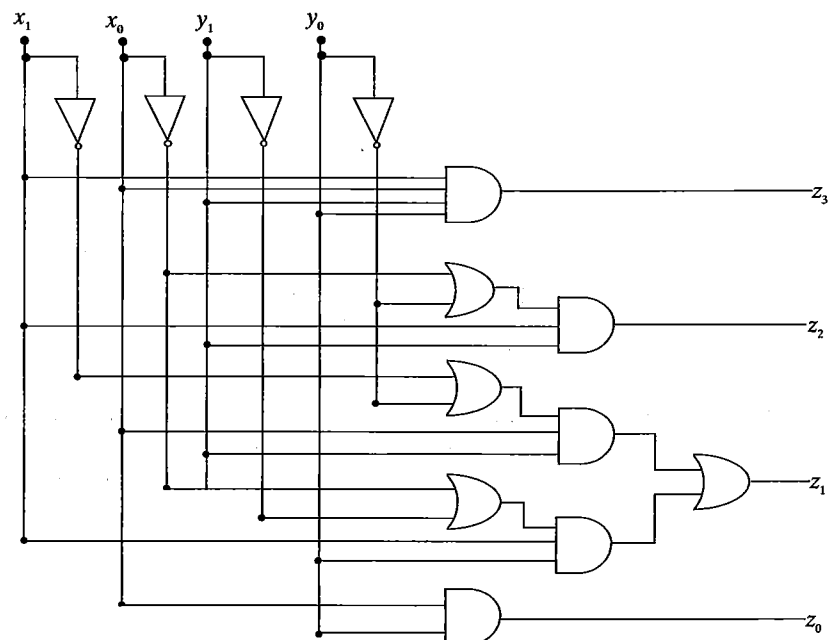


Figura 3.26.

6.5. Regulación de una piscina

Se trata de diseñar un sistema de regulación de la temperatura y del nivel del agua en una piscina. El sistema recibirá la información de tres sensores colocados dentro de la piscina: un termómetro y dos detectores de nivel. Y deberá actuar sobre un calefactor eléctrico y sobre dos válvulas, una que permite desalojar agua

y otra que permite añadir agua fría. Las especificaciones funcionales son las dadas en la siguiente tabla:

<i>Información recibida</i>		<i>Acción a tomar</i>
<i>Temperatura</i>	<i>Nivel</i>	
Normal	Normal	Ninguna
Normal	Bajo	Añadir agua
Normal	Alto	Sacar agua
Baja	Normal	Calentar
Alta	Normal	Añadir agua
Baja	Bajo	Añadir agua y calentar
Baja	Alto	Sacar agua y calentar
Alta	Bajo	Añadir agua
Alta	Alto	Añadir y sacar agua al mismo tiempo

Se supondrá que el termómetro tiene dos terminales de salida, cada uno de los cuales puede estar en uno de entre dos niveles de tensión (a los que daremos los niveles lógicos 0 y 1). Si la temperatura es demasiado alta, el primer terminal, T_1 , está en el nivel 1 (y el segundo, T_2 , en el 0), si es demasiado baja ocurre lo contrario, y si está entre los límites prefijados ("temperatura normal") ambos son 0.

Los detectores del nivel del agua dan también niveles 0 y 1. Uno de ellos, N_1 , da 1 si el nivel es demasiado alto, y el otro, N_2 , da 1 si es demasiado bajo (de modo que 0 en ambos indica "nivel normal").

En cuanto a la salida, V_1 es la válvula para sacar agua y V_2 es la válvula para añadir agua. El nivel lógico 0 corresponde a la activación de la válvula, y el 1 a la desactivación. Análogamente, para que el calefactor, o resistencia, (R) caliente se dará 1, y para que no actúe, 0.

La parte lógica del sistema será un circuito con cuatro entradas (T_1, T_2, N_1, N_2) y tres salidas (V_1, V_2, R). Las tablas de verdad se obtienen inmediatamente de las especificaciones. Hay siete combinaciones de valores de entrada que son imposibles (las que corresponden a $T_1 = T_2 = 0$, y las que corresponden a $N_1 = N_2 = 0$), por lo que las funciones de V_1, V_2 y R están incompletamente especificadas. Aprovecharemos este hecho, del mismo modo que en el ejemplo 6.3, para minimizar tales funciones. Si el lector sigue los pasos necesarios llegará al siguiente resultado:

$$V_1 = N_1; V_2 = T_1 + N_2; R = T_2$$

Y el circuito (en lo que respecta al nivel lógico) sólo necesita de una puerta "OR", como indica la figura 3.27.

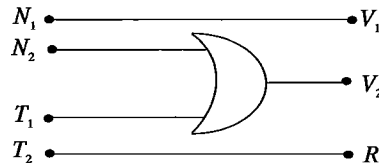


Figura 3.27.

7. La segunda forma canónica y la forma mínima en producto de sumas

7.1. La segunda forma canónica

En el apartado 4.3.5 hemos visto cómo cada clase de equivalencia en el conjunto de formas booleanas construidas con n variables puede representarse por una forma canónica formada por sumas de productos canónicos. Existen, como vamos a ver, otras posibles formas canónicas para representar de manera única a las clases de equivalencia, por lo que, para diferenciarlas, llamaremos a la que ya conocemos *primera forma canónica* o *forma canónica en suma de productos*. A los productos canónicos se les llama también *minitérminos*.

Definición 7.1.1. Llamaremos *suma canónica*, o *maxitérmino*, a toda forma booleana compuesta por la suma de todas las variables complementadas o no:

$$S_i = l_1 + l_2 + \dots + l_n = \sum_{j=1}^n l_j$$

(donde " Σ " representa la suma booleana).

Definición 7.1.2. Llamaremos *segunda forma canónica* (o *forma canónica en producto de sumas*) a toda forma booleana compuesta por un producto de maxitérminos diferentes entre sí.

Teorema 7.1.3. Toda clase de equivalencia en C_n puede representarse mediante su segunda forma canónica, que es única para esa clase de equivalencia.

Demostración:

Sea $c_i \in C_n$ una clase de equivalencia cualquiera, y consideremos su complementaria, $\overline{c_i}$. Por el teorema 4.3.5.3 sabemos que c_i tiene una representación única en la primera forma canónica:

$$\bar{c}_i = \sum_{j=1}^n (\prod l_j)$$

Aplicando los teoremas de de Morgan, obtenemos:

$$c_i = \overline{\left[\sum_{j=1}^n \left(\prod l_j \right) \right]} = \prod \left[\overline{\left(\prod l_j \right)} \right] = \prod \left(\sum_{j=1}^n \bar{l}_j \right)$$

que será la representación de c_i en la segunda forma canónica. Y como \bar{c}_i es única para c_i , esta representación también será única.

Una notación abreviada para la segunda forma canónica se obtiene asignando, a cada maxitérmino un número decimal. Por ejemplo, $x + \bar{y} + \bar{z}$ sólo vale "0" para la combinación 011 (3 en decimal), $\bar{x} + y + z$ sólo vale "0" para 101 (5), y la representación de $(x + \bar{y} + z) \cdot (\bar{x} + y + z)$ será: $\Pi(3,5)$.

Ejemplo 7.1.4. La función de conmutación correspondiente a la disyunción exclusiva ("OR" exclusivo) viene dada por esta tabla de verdad (en la que hemos incluido también los valores de \bar{f}):

x	y	f	\bar{f}
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Considerando los productos canónicos que corresponden a los "1" de f obtenemos la primera forma canónica:

$$f = \bar{x} \cdot y + x \cdot \bar{y}$$

Y considerando los que corresponden a los "1" de \bar{f} ("0" de f) obtenemos la primera forma canónica de \bar{f} :

$$\bar{f} = \bar{x} \cdot \bar{y} + x \cdot y$$

De aquí,

$$f = \overline{\bar{x} \cdot \bar{y} + x \cdot y} = (\overline{\bar{x} \cdot \bar{y}}) \cdot (\overline{x \cdot y}) = (x + y) \cdot (\bar{x} + \bar{y})$$

que será la segunda forma canónica de f (en notación abreviada, $\Pi(0,3)$).

Ejemplo 7.1.5. Para los ejemplos 3.1 y 3.2, fijándonos en los "0" de la tabla de verdad, resulta:

$$\bar{f} = \bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z}$$

y, de aquí, la segunda forma canónica:

$$f = (x + y + z) \cdot (x + y + \bar{z}) \cdot (x + \bar{y} + z) = \Pi(0, 1, 2)$$

Ejemplo 7.1.6. Análogamente, para los ejemplos 3.5, 3.6 y 3.7:

$$\bar{f} = \bar{x} \cdot \bar{y} \cdot \bar{z} + \bar{x} \cdot \bar{y} \cdot z + \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z}$$

$$f = (x + y + z) \cdot (x + y + \bar{z}) \cdot (x + \bar{y} + z) \cdot (\bar{x} + y + z) = \Pi(0, 1, 2, 4)$$

Ejemplo 7.1.7. Para los ejemplos 3.3 y 3.4,

$$\bar{f} = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4 + \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot x_4 + \bar{x}_1 \cdot x_2 \cdot \bar{x}_3 \cdot \bar{x}_4 + \bar{x}_1 \cdot x_2 \cdot \bar{x}_3 \cdot x_4 + x_1 \cdot x_2 \cdot \bar{x}_3 \cdot \bar{x}_4$$

$$f = (x_1 + x_2 + x_3 + x_4) \cdot (x_1 + x_2 + x_3 + \bar{x}_4) \cdot (x_1 + \bar{x}_2 + x_3 + x_4) \cdot (x_1 + \bar{x}_2 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + \bar{x}_2 + x_3 + x_4) = \Pi(0, 1, 4, 5, 12)$$

7.2. La forma mínima en producto de sumas

Los dos principios que permiten la simplificación de formas booleanas expresadas en producto de sumas son duales de los que veíamos en el apartado 5.1:

a) *Reducción de términos adyacentes:* $(A + x) \cdot (A + \bar{x}) \equiv A$

Obsérvese que esta "reducción" se corresponde exactamente con la "regla de resolución" de la lógica (capítulo 2, apartado 5.7).

b) *Idempotencia:*

$$A \cdot A \equiv A$$

Para el mismo ejemplo que analizábamos en el apartado 5.1, la segunda forma canónica es exactamente la que hemos visto en el ejemplo 7.1.6. Podemos observar que el primer maxitérmino es adyacente a los otros tres. Si lo descomponemos en el producto de tres iguales y reducimos cada uno de ellos con cada uno de los otros tres, resulta:

$$f = (x + y) \cdot (x + z) \cdot (y + z)$$

que es la forma mínima en producto de sumas. Aplicando la propiedad distributiva, podemos escribir otras formas equivalentes:

$$(x + y) \cdot (z + x \cdot y) \equiv (y + z) \cdot (x + y \cdot z) \equiv (x + z) \cdot (y + x \cdot z)$$

A efectos prácticos, y utilizando el método de las tablas de Karnaugh, lo más cómodo es minimizar \bar{f} (agrupando los "0" de la tabla en lugar de los "1") y luego aplicar los teoremas de de Morgan. El circuito resultante mediante este procedimiento es, generalmente, el mismo obtenido a partir de la forma mínima en suma de productos, o su dual (cambiando las puertas "OR" por "AND" y viceversa). Pero a veces, cuando la función está incompletamente especificada, el circuito puede ser más sencillo (o más complicado). Como ejercicio, el lector puede probar a hacer las minimizaciones correspondientes a los ejemplos 5.2.1 a 5.2.8 y comparar los circuitos que se obtienen de una manera y de la otra. Veamos cuáles serían los circuitos alternativos para algunos de los "ejemplos de aplicación" del apartado 6.

Ejemplo 7.2.1. Si agrupamos los "0" en la tabla de Karnaugh del ejemplo 6.1 (fig. 3.28(a)) resulta:

$$\bar{S} = \bar{A} \cdot \bar{D} + \bar{A} \cdot \bar{C} + \bar{A} \cdot \bar{B} + \bar{B} \cdot \bar{C} \cdot \bar{D}$$

$$S = (A + D) \cdot (A + C) \cdot (A + B) \cdot (B + C + D)$$

O, aplicando la propiedad distributiva:

$$S = (A + B \cdot C \cdot D) \cdot (B + C + D)$$

El circuito (fig 3.28(b)) es igual que el de la figura 3.20(b), cambiando los tipos de puertas.

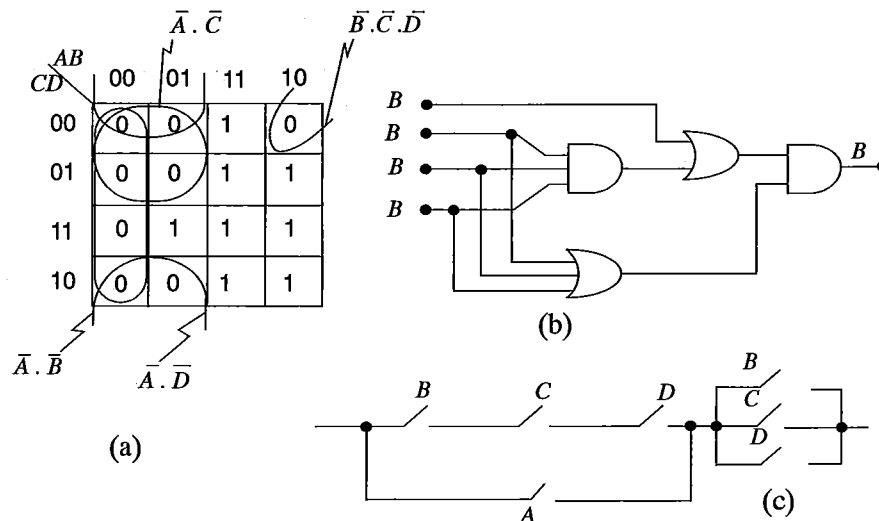


Figura 3.28.

Ejemplo 7.2.2. Para el caso del ejemplo 6.2, obtenemos ahora (figura 3.29):

$$\bar{S} = \bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C} \cdot \bar{D}$$

$$S = (A + B) \cdot (A + C + D) = A + B(C + D)$$

Forma que es la misma a la que habíamos llegado por el otro procedimiento, por lo que el circuito sería el mismo (figura 3.21(b)).

	00	01	11	10
00	0	0	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

Figura 3.29.

Ejemplo 7.2.3. La función de conmutación de la salida r del sumador binario es exactamente el ejemplo que hemos considerado al principio de este apartado (llamando z a r'). Una de las formas minimizadas era:

$$r = (x + y) \cdot (r' + x \cdot y)$$

En cuanto a s , si tomamos r como entrada adicional, resulta (figura 3.30):

$$\bar{s} = \bar{r}' \cdot r + \bar{x} \cdot r + \bar{y} \cdot r + \bar{x} \cdot \bar{y} \cdot \bar{r}'$$

$$s = (r' + \bar{r}) \cdot (x + \bar{r}) \cdot (y + \bar{r}) \cdot (x + y + r')$$

$$= (\bar{r} + x \cdot y \cdot r') \cdot (x + y + r')$$

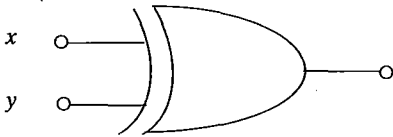
Y el circuito es el dual del de la figura 3.24.

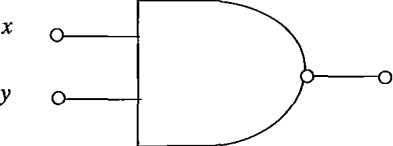
	00	01	11	10
00	0	1	0	1
01	0	0	0	0
11	0	0	1	0
10	1	0	0	0

Figura 3.30.

8. Otras puertas

Igual que en lógica de proposiciones, donde teníamos dieciséis conectivas binarias (entre dos variables proposicionales), también podemos definir dieciséis operaciones distintas entre dos variables lógicas, a las que corresponderían otras tantas puertas de dos entradas, aunque cuatro de ellas no tienen sentido: las que dan siempre "0" o "1" a la salida (las tautologías y contradicciones de la lógica) y las que dan el valor de una de las entradas independientemente del que tome la otra. Ahora bien, las más utilizadas, aparte de las ya vistas ("NOT", "OR" y "AND") son "ORX" ("OR" exclusivo), "NAND" y "NOR", cuyas tablas de verdad son las de la figura 3.31. Para simbolizar las operaciones que realizan estas puertas utilizaremos los símbolos " \oplus ", " \uparrow " y " \downarrow ", respectivamente (que son los mismos que ya habíamos visto en el capítulo 2 para las correspondientes conectivas).

OR EXCLUSIVO		
x		
y		
x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

NAND		
x		
y		
x	y	f
0	0	1
0	1	1
1	0	1
1	1	0

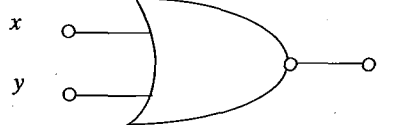
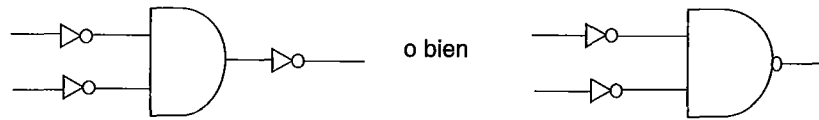
NOR		
x		
y		
x	y	f
0	0	1
0	1	0
1	0	0
1	1	0

Figura 3.31.

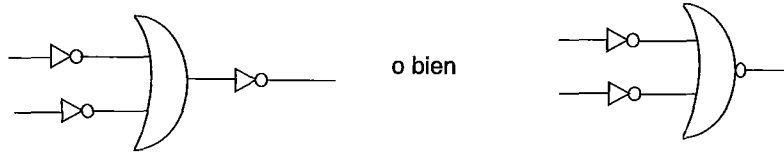
Y del mismo modo que unas conectivas pueden expresarse en función de otras (capítulo 2, apartado 3.6), también la función que realiza una puerta puede realizarse con un circuito formado por otras. Por ejemplo, la suma lógica con "AND" y "NOT" (figura 3.32(a)), el producto lógico con "OR" y "NOT" (figura 3.32(b)), "NAND" y "NOR" con "AND" y "NOT" o "OR" y "NOT" (figura

3.32(c) y (d)), etc. Obsérvense, en la figura, las representaciones alternativas para "NAND" y "NOR": la misma función se realiza complementando la salida de una puerta "AND" que complementando las entradas de una "OR", y viceversa.

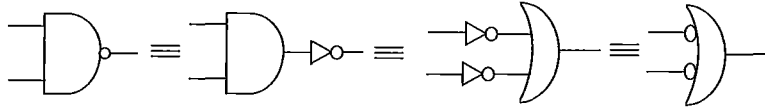
$$(a) x + y = \overline{\overline{x} \cdot \overline{y}} = \overline{x} \downarrow \overline{y}$$



$$(b) x \cdot y = \overline{\overline{x} + \overline{y}} = \overline{x} \uparrow \overline{y}$$



$$(c) x \downarrow y = \overline{x \cdot y} = \overline{x} + \overline{y}$$



$$(d) x \uparrow y = \overline{x + y} = \overline{x} \cdot \overline{y}$$

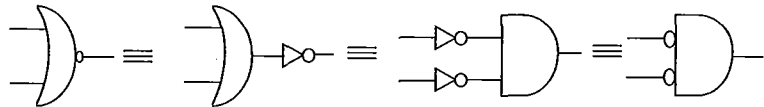
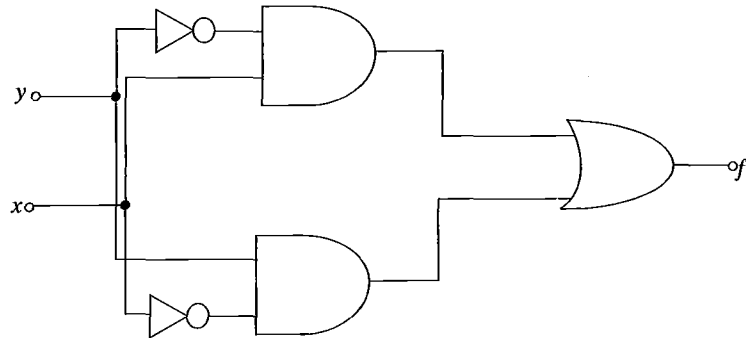


Figura 3.32.

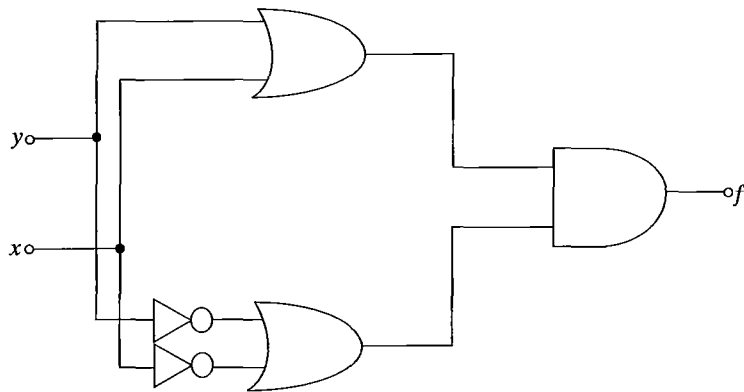
De acuerdo con las dos formas canónicas vistas en el ejemplo 7.1.4 (que no pueden reducirse), la puerta "ORX" puede realizarse con puertas "NOT", "OR" y "AND" con el circuito de la figura 3.33(a) o con el de la figura 3.33(b). La disponibilidad directa de puertas "ORX" permite realizar de forma más simple algunos circuitos. Por ejemplo, la salida z_1 del ejemplo 6.4 puede también escribirse:

$$z_1 = (x_0 \cdot y_1) \oplus (x_1 \cdot y_0)$$

Por tanto, si puede utilizarse una puerta "ORX" el circuito se simplifica algo con relación al de la figura 3.26.



(a) $f = x_1 \cdot \bar{x}_2 + \bar{x}_1 \cdot x_2$



(b) $f = (x + y) \cdot (\bar{x} + \bar{y})$

Figura 3.33.

9. Circuitos con puertas "NAND" y "NOR"

9.1. "NAND" y "NOR" como operaciones completas

Por razones tecnológicas, las puertas "NAND" y "NOR" se utilizan mucho en el diseño de circuitos. Estas puertas puede tener más de dos entradas:

$$x|y|z\dots = \overline{x \cdot y \cdot z\dots} = \bar{x} + \bar{y} + \bar{z} + \dots$$

$$x \downarrow y \downarrow z \downarrow \dots = \overline{x + y + z + \dots} = \bar{x} \cdot \bar{y} \cdot \bar{z} \cdot \dots$$

Tanto NAND como NOR tienen la propiedad de ser *operaciones completas*; esto quiere decir que cualquier función de conmutación puede representarse utilizando sólo la operación NAND o utilizando sólo la operación NOR. Para demostrarlo basta con ver que las tres operaciones básicas, complementación, producto y suma, pueden realizarse con ellas:

$$a) \bar{x} = \overline{x \cdot x} = x | x$$

$$\bar{x} = \overline{x + x} = x \downarrow x$$

$$b) x \cdot y = \overline{\overline{x \cdot y}} = \overline{\bar{x} | \bar{y}} = (x | y) | (x | y)$$

$$x \cdot y = \overline{x + y} = \bar{x} \downarrow \bar{y} = (x \downarrow x) \downarrow (x \downarrow y)$$

$$c) x + y = \overline{\overline{x + y}} = \overline{\bar{x} | \bar{y}} = (x | x) | (y | y)$$

$$x + y = \overline{\overline{x + y}} = \overline{(x \downarrow y)} = (x \downarrow y) \downarrow (x \downarrow y)$$

Es preciso prestar atención a los paréntesis, ya que, a diferencia de la suma y el producto, NAND y NOR *no* son asociativas:

$$(x | y) | z \neq x | (y | z) \neq x | y | z$$

$$(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z) \neq x \downarrow y \downarrow z$$

(Compruébese por medio de las respectivas tablas de verdad).

9.2. La tercera forma canónica

La tercera forma canónica (sólo con NAND) se obtiene directamente de la primera mediante aplicación del

Teorema 9.2.1:

$$(x_1 \cdot x_2 \cdot \dots \cdot x_n) + (y_1 \cdot y_2 \cdot \dots \cdot y_m) + \dots + (z_1 \cdot z_2 \cdot \dots \cdot z_h)$$

$$= (x_1 | x_2 | \dots | x_n) | (y_1 | y_2 | \dots | y_m) | \dots | (z_1 | z_2 | \dots | z_h)$$

(Para demostrarlo basta con aplicar dos veces las leyes de De Morgan).

9.3. La forma mínima sólo con NAND

Se obtiene aplicando el Teorema 9.2.1 a la forma mínima en suma de productos.

Ejemplo (corresponde al Ejemplo 5.2.6):

$$\begin{aligned} f &= x_3 + x_1 \cdot x_4 + x_1 \cdot \bar{x}_2 = (x_3 | x_3) | (x_1 | x_4) | (x_1 | \bar{x}_2) = \\ &= (x_3 | x_3) | (x_1 | x_4) | (x_1 | (x_2 | x_2)) \end{aligned}$$

(Obsérvese que cuando uno de los productos consta de una sola variable, en este caso x_3 , este producto puede representarse como $x_3 \cdot x_3$, y de ahí que al aplicar el teorema pongamos $x_3 | x_3$ (es decir, \bar{x}_3), y no x_3).

9.4. La cuarta forma canónica

Se obtiene de la segunda aplicando el

Teorema 9.4.1:

$$\begin{aligned} &(x_1 + x_2 + \dots + x_n) \cdot (y_1 + y_2 + \dots + y_m) \cdot \dots \cdot (z_1 + z_2 + \dots + z_h) \\ &= (x_1 \downarrow x_2 \downarrow \dots \downarrow x_n) \downarrow (y_1 \downarrow y_2 \downarrow \dots \downarrow y_m \downarrow \dots \downarrow (z_1 \downarrow z_2 \downarrow \dots \downarrow z_h)) \end{aligned}$$

(Se demuestra igualmente con las leyes de De Morgan).

9.5. La forma mínima sólo con NOR

Si aplicamos el Teorema 9.4.1 a la forma mínima en producto de sumas, obtenemos como resultado final, una forma mínima con operaciones NOR solamente.

Ejemplo:

$$\begin{aligned} f &= x \cdot (\bar{x} + y) \cdot (y + \bar{z}) = (x \downarrow x) \downarrow (\bar{x} \downarrow y) \downarrow (y \downarrow \bar{z}) = \\ &= (x \downarrow x) \downarrow ((x \downarrow x) \downarrow y) \downarrow (y \downarrow (z \downarrow z)) \end{aligned}$$

(Aquí podemos hacer una observación similar a la del ejemplo anterior: si una de las sumas sólo tiene un sumando, x en este caso, como $x = x + x$, al aplicar el teorema ponemos $x \downarrow x = \bar{x}$, en lugar de x).

10. Resumen

Las funciones de conmutación y las formas booleanas son dos tipos completamente distintos de modelos para los circuitos lógicos combinacionales. Una

función de conmutación es un *modelo funcional*: indica "lo que hace" el circuito, desde el punto de vista de cuáles son sus "respuestas" (valores lógicos de la salida) para los posibles "estímulos" (valores lógicos de las entradas). Por el contrario, una *forma booleana* es un *modelo estructural*: indica cómo se conectan los componentes (puertas) del circuito. A cada circuito que sólo tenga una salida le corresponde una forma booleana, y viceversa. Pero a una misma función de conmutación le corresponden infinitud de circuitos (y, por tanto, de formas booleanas).

Frecuentemente, de las especificaciones verbales se llega inmediatamente a las funciones de conmutación (una para cada salida), expresadas como tablas de verdad. Y el problema que se plantea es el de elegir de entre todos los circuitos (formas booleanas) posibles el más sencillo (forma booleana con menos operaciones)². A la solución de este problema general ha ido encaminado todo el desarrollo de este capítulo, que podemos resumir así:

- a) El conjunto de funciones de conmutación de orden n , junto con unas operaciones de suma, producto y complementación adecuadamente definidas, constituye un álgebra de Boole, $\langle F_n, +, \cdot, \bar{} \rangle$.
- b) Dentro del conjunto de formas booleanas de n variables, B_n , hemos definido una relación de equivalencia (que también es una relación de equivalencia entre los circuitos con n entradas y una salida). Esta relación establece una partición de B_n en un conjunto de clases de equivalencia, C_n .
- c) El conjunto de clases de equivalencia entre formas booleanas de n variables, junto con las operaciones de suma producto y complementación definidas en él, constituye un álgebra de Boole, $\langle C_n, +, \cdot, \bar{} \rangle$.
- d) Las álgebras de Boole $\langle F_n, +, \cdot, \bar{} \rangle$ y $\langle C_n, +, \cdot, \bar{} \rangle$ son isomorfas, porque tienen el mismo número de elementos ($\text{card}(F_n) = \text{card}(C_n) = 2^{2^n}$). Por tanto, a cada función de conmutación de orden n le corresponde biunívocamente una clase de equivalencia entre formas booleanas de n variables, y, por tanto, un conjunto (infinito) de circuitos equivalentes. Esta doble correspondencia se ilustra en la figura 3.34.
- e) Hemos definido las *formas canónicas*, formas booleanas que permiten representar de manera única a las clases de equivalencia, y que se obtienen inmediatamente a la vista de la función de conmutación.

² Esta es, desde luego, una simplificación del problema general. No siempre el circuito con menos puertas es el de menor coste de realización, y esto es especialmente cierto desde la aparición de los circuitos integrados.

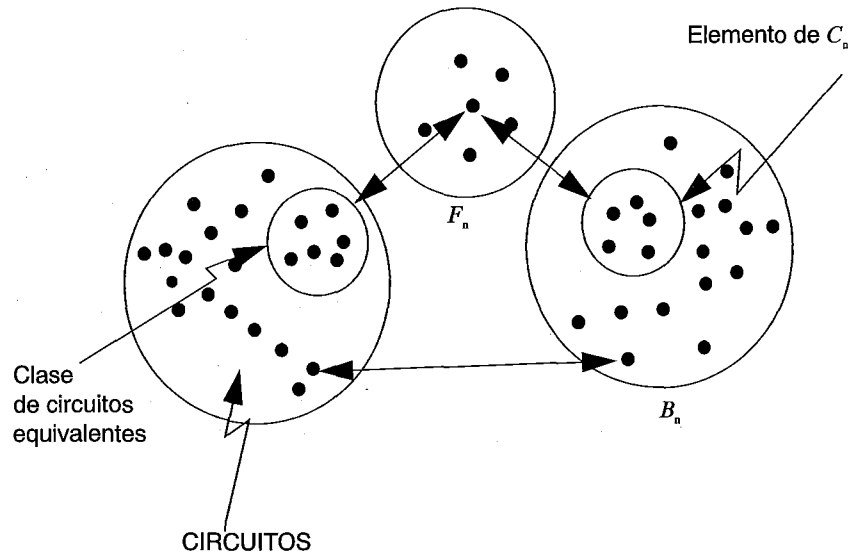


Figura 3.34.

- f) Desde un punto de vista práctico, lo que interesa es, dada una función de conmutación, o una forma canónica, o una forma booleana cualquiera, encontrar la forma booleana que esté en la misma clase de equivalencia y que tenga el menor número posible de operaciones (que corresponderá a un circuito con el menor número posible de puertas). Es el problema de la minimización, para cuya solución hemos estudiado el método más sencillo: el de las tablas de Karnaugh.
- g) También hemos analizado algunos casos en los que la función de conmutación está *incompletamente especificada*, lo que se aprovecha asimismo para minimizar el circuito.
- h) Finalmente, hemos visto cómo a través de unas sencillas transformaciones cualquier forma booleana puede expresarse mediante otra equivalente que sólo contiene operaciones "NAND" o sólo operaciones "NOR", lo que permite diseñar circuitos con uno u otro de esos tipos de puertas exclusivamente.

11. Notas histórica y bibliográfica

Fue Shannon (1938) el primero en aplicar el álgebra de Boole como modelo matemático para los circuitos de conmutación (cuyos componentes básicos, en esa época, eran de tecnología electromecánica). En los años siguientes, diversos

autores, entre ellos el propio Shannon (1949), desarrollaron la teoría. El método gráfico para la minimización lo introdujo Veitch (1952) y lo perfeccionó Karnaugh (1953).

La tecnología de realización de circuitos digitales ha evolucionado muy rápidamente en los últimos años, pero las técnicas básicas de diseño, independientes de esa tecnología, son prácticamente las mismas de los años 60. Por eso, libros de esa época, como los de Bartee (1960), Bartee *et al.* (1962), Harrison (1965), Kohavi (1970) y Oberman (1970) son aún referencias válidas (aunque difíciles de encontrar, si no es en bibliotecas que tengan el buen sentido de conservar ejemplares). Algunos de ellos se han ido actualizando para recoger los avances tecnológicos. Por ejemplo, el primero, tras múltiples ediciones, incluye ahora (Bartee, 1991) descripciones de circuitos integrados y de microprocesadores de dieciséis y treinta y dos bits, además de otros temas propios de la arquitectura de ordenadores: tecnologías de memorias, de buses, de dispositivos periféricos, etc.

Otros libros más recientes sobre el diseño de circuitos lógicos son los de Mandado (1991), Mano (1991), Wilkinson y Makki (1992), Wakerky (1993), Sanders (1993), Mano y Kime (1994) y Tocci (1995). La mayoría de los textos sobre arquitectura de ordenadores (por ejemplo, Patterson y Hennessy, 1994 o Fernández, 1994) incluyen un apéndice con lo esencial de los sistemas lógicos. En el campo muy especializado del diseño de circuitos lógicos con tecnología de integración de muy alta densidad (VLSI: Very Large Scale Integration) hay que citar el texto ya clásico de Mead y Conway (1980), o los más recientes de Wolf (1994) y Puckwell y Eshraghian (1994).

12. Ejercicios

- 12.1. Considérese el ejemplo 5.7.3 del capítulo 2. Cada cláusula se corresponde, en el lenguaje de las formas booleanas, con una suma de variables. (Concretamente, P_2 es una suma canónica). Representar en una tabla de Karnaugh las evaluaciones de la conjunción de las tres cláusulas, y ver cómo las dos aplicaciones de la regla de resolución que hacíamos allí se corresponden con reducciones de términos adyacentes, y tienen su representación gráfica como agrupaciones de "0".
- 12.2. Diseñar un circuito con cuatro entradas, x_0 a x_3 , que representan los bits componentes de los números 0 a 15 expresados en binario (0000 a 1111), y una salida, f , que sólo debe tomar el valor "1" en el caso de que el número presentado a la entrada sea múltiplo de 3. En éste, como en los siguientes ejercicios, se diseñarán cuatro circuitos minimizados: utilizando puertas "NOT", "AND" y "OR", por los dos métodos (forma mínima en suma de productos y forma mínima en producto de sumas), utilizando sólo puertas "AND" y utilizando sólo puertas "OR".

- 12.3.** El mismo ejercicio anterior, considerando que los números que se presentan a la entrada están siempre comprendidos entre 0 y 9.
- 12.4.** Los dígitos decimales pueden codificarse en binario como se hace en el anterior ejercicio: asignando a cada uno el conjunto de cuatro bits que corresponde a su escritura en el sistema de numeración de base 2. Pero éste, que se llama código BCD "natural", no es el único código BCD (Binary Coded Decimal) posible: como sólo necesitamos diez de los dieciséis símbolos que pueden codificarse con cuatro bits, hay, realmente, $V_{16,10} = 16!/6! \approx 2.9 \cdot 10^{10}$ códigos BCD diferentes. Tres de los más utilizados (incluyendo el "8-4-2-1", que es el también llamado "natural") son:

	8-4-2-1	2-4-2-1	exceso de 3
0	0000	0000	0011
1	0001	0001	0100
2	0010	0010	0101
3	0011	0011	0110
4	0100	0100	0111
5	0101	0101	1000
6	0110	0110	1001
7	0111	0111	1010
8	1000	1110	1011
9	1001	1111	1100

Diseñar un circuito para pasar del "8-4-2-1" a uno de los otros dos. El circuito tendrá cinco entradas: las cuatro correspondientes a los bits del código fuente y una entrada de control, C . Si $C=0$ entonces sus cuatro salidas deben dar los bits del código "2-4-2-1" que correspondan, y si $C=1$ en la salida se deberá obtener el código "exceso de 3".

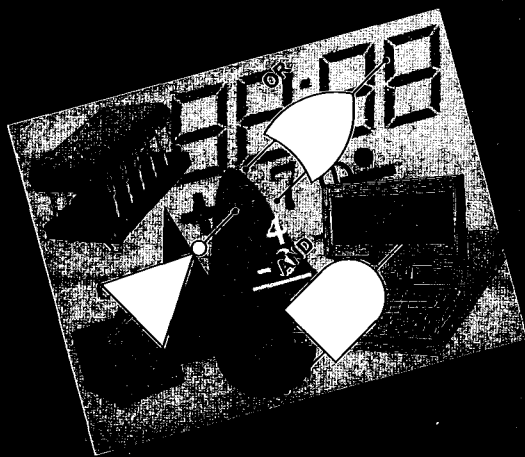
- 12.5.** Diseñar un circuito que permita obtener el cuadrado de cualquier número comprendido entre 0 y 7.
- 12.6.** Un codificador es un circuito con 2^n entradas y n salidas que genera a la salida el código correspondiente a la entrada cuyo valor es "1" (se supone que en cada momento sólo una de las entradas puede tomar el

valor "1"). Diseñar un codificador con cuatro entradas (y dos salidas). Generalizar el diseño para ocho entradas (tres salidas).

- 12.7.** Un multiplexor es un circuito con 2^n entradas de información binaria y n entradas de selección y una sola salida. Esta salida toma el valor de la entrada de información que haya sido seleccionada por las entradas de selección. Diseñar un multiplexor con $n=3$.

Fundamentos de informática

4



4

Lógica de predicados de primer orden

1. Introducción

1.1. Variables y constantes

En el capítulo 2 analizábamos un ejemplo (el 1.4.4) en el que parecía natural introducir la idea de variable para referirnos a un miembro de un colectivo. Hay razonamientos cuyo análisis es imposible si no es formalizando esa idea. Volvamos, por ejemplo, al clásico:

todos los hombres son mortales

Sócrates es un hombre

luego Sócrates es mortal

Si tratamos de formalizar este razonamiento en lógica de proposiciones, para la primera premisa podríamos escribir la sentencia " $h \rightarrow m$ ". La segunda premisa, al ser un simple enunciado declarativo, se formalizaría como una simple variable proposicional, " s ". Y de tales dos premisas es imposible inferir ninguna conclusión.

Lo que ocurre en este ejemplo es que en la primera premisa estamos diciendo algo de *todos* los miembros de un colectivo, que, por tanto, es válido para uno *cualquiera* de ellos, al cual representaremos con una *variable*. En la segunda, hablamos de un valor determinado de esa variable, de una *constante*. Y la relación de deducibilidad que aplica el razonamiento es un paso de lo general a lo particular: lo que se dice de todos los miembros de un colectivo es válido para uno cualquiera de ellos. Para expresarla es preciso entrar en la composición de los enunciados, cosa que no puede hacerse en lógica de proposiciones, en la que todo enunciado declarativo simple se representa como una variable proposicional.

1.2. Propiedades y relaciones

En el ejemplo anterior hablamos de *propiedades* de individuos: la propiedad de ser hombre y la propiedad de ser mortal. Y estas propiedades las aplicábamos a un individuo cualquiera (una variable: "hombre") o a uno concreto (una constante: "Sócrates"). Otros enunciados se refieren a *relaciones* entre individuos. Por ejemplo, en "Juan ama a María" establecemos una relación unidireccional ("ama a") entre dos individuos determinados (dos constantes), en "algunos hombres aman en secreto a Ana Belén" la relación es entre una variable (hombre) y una constante (Ana Belén), y en "todos los hombres aman a alguien" es entre dos variables.

Pueden establecerse relaciones entre un número cualquiera de individuos. Así, cuando decimos "Juan regala flores a María" establecemos una relación ternaria ("regala") entre tres constantes.

1.3. Predicados y fórmulas atómicas

Un predicado es la formalización de una propiedad o de una relación. Para propiedades, tendremos *predicados monádicos*, que tienen un solo argumento (constante o variable). Seguiremos la notación de representar los predicados con letras mayúsculas, y las constantes con las minúsculas a, b, c, \dots , reservando x, y, z para las variables. El argumento se pondrá entre paréntesis a continuación del predicado. Por ejemplo:

"Sócrates es mortal": $M(s)$

"alguien es mortal" : $M(x)$

Para relaciones entre dos individuos tendremos los *predicados diádicos*:

"Juan ama a María": $A(j, m)$

"alguien ama a Ana": $A(x,a)$

"alguien ama a otro": $A(x,y)$

Y, en general, tendremos predicados poliádicos:

"Juan regala flores a María": $R(j,m,f)$

(Gramaticalmente, esta frase tendría un sujeto, "Juan", y un predicado, formado por el verbo y los complementos. Pero en la definición que se da en lógica el predicado es sólo el verbo, y los complementos, junto con el sujeto, son los argumentos de ese predicado).

La construcción formada por un predicado seguido de sus argumentos se llama *fórmula atómica*. Obsérvese que las fórmulas atómicas representan a enunciados declarativos; corresponden, pues, a las variables proposicionales, en cuya composición ahora estamos entrando.

1.4. Sentencias abiertas y cerradas

Igual que en lógica de proposiciones construíamos sentencias enlazando variables proposicionales con las conectivas, aquí lo haremos enlazando fórmulas atómicas con las mismas conectivas. (Una simple fórmula atómica también será una sentencia).

Así como en lógica de proposiciones, dada una interpretación, podíamos evaluar las variables proposicionales como verdaderas o falsas, y, a partir de una determinada evaluación, calcular la evaluación de la sentencia, ahora no siempre podemos hacerlo. Porque si tenemos, por ejemplo, un predicado monádico, su valor de verdad o falsedad normalmente dependerá del valor que tome la variable: si digo " x es español", o, formalmente, $E(x)$, dependiendo de quién sea x el resultado será verdadero o no.

En general, tendremos *sentencias abiertas*, que son aquellas que, al depender de variables, no se les puede atribuir un valor de verdad o falsedad, y *sentencias cerradas*, en las que tal atribución es posible.

Un modo obvio de *cerrar* una sentencia abierta consiste en fijar valores para las variables que intervienen en ella. En el último ejemplo, si hacemos $x = \text{Picasso}$, la sentencia (en este caso, fórmula atómica) $E(x)$ es verdadera, y si $x = \text{Matisse}$, $E(x)$ es falsa.

Pero las sentencias pueden ser también verdaderas o falsas sin que tengan que referirse exclusivamente a constantes: "todos los hombres son mortales" es una sentencia verdadera, y, por tanto, cerrada. Ello es así porque lo que se predica es algo sobre una variable, pero que es válido para *todos* los valores de la variable. Por tanto, a la sentencia $H(x) \rightarrow M(x)$ (si alguien, x , es hombre entonces es mortal) hay que añadirle algo que exprese su validez para cualquier valor de x , cerrando de ese modo la sentencia. Ese algo se llama "cuantificador universal".

1.5. Cuantificadores

Representaremos al *cuantificador universal* con el símbolo " \forall " seguido de la variable que se cuantifica. Para delimitar el *alcance* de la cuantificación, pondremos la sentencia cuya variable se cuantifica entre paréntesis. Por ejemplo,

$$(\forall x) (H(x) \rightarrow M(x))$$

es una sentencia cerrada, que formaliza la frase "para todo x , si x tiene la propiedad H , entonces tiene la propiedad M ", mientras que

$$(\forall x) (H(x)) \rightarrow M(x)$$

sería una sentencia abierta ("si todos los x tienen la propiedad H , entonces x , cualquiera, tiene la propiedad M "), lo mismo que

$$(\forall x) (H(x) \rightarrow M(y))$$

("para todos los x , si x tiene la propiedad H , entonces y tiene la propiedad M ", que viene a ser lo mismo de antes, supuesto que x e y tienen un rango común).

El otro cuantificador, también conocido del lenguaje matemático, es el *cuantificador existencial*, que representaremos con el símbolo " \exists ", y que se lee "existe un... tal que...".

En realidad, bastaría con uno solo de los dos cuantificadores, y si usamos los dos es por comodidad (por acercar el lenguaje lógico al lenguaje natural). En efecto, decir que todos los individuos en consideración tienen cierta propiedad es lo mismo que decir que no es cierto que exista algún individuo que no tenga esa propiedad:

$$(\forall x) (P(x)) \equiv \neg(\exists x) (\neg P(x))$$

Y decir que existe un x (al menos) que tenga cierta propiedad es lo mismo que decir que no es cierto que para todos los individuos la propiedad no se da:

$$(\exists x) (P(x)) \equiv \neg(\forall x) (\neg P(x))$$

1.6. Evaluaciones binarias

La función de interpretación es más compleja en lógica de predicados que en lógica de proposiciones. En efecto, la conceptualización (que en lógica de proposiciones era, simplemente, un conjunto de proposiciones) ahora contiene elementos de distinta naturaleza: individuos (u objetos), propiedades, relaciones y funciones, y la interpretación consiste en asignar elementos de esa conceptualización a los símbolos utilizados en las sentencias. Definiremos matemáticamente esta función en el apartado 3. De momento, para esta introducción,

basta saber que, establecida una interpretación, existirá una evaluación, que en todo este capítulo supondremos binaria, para cada fórmula atómica con argumentos constantes y para cada sentencia cerrada.

Ahora bien, como ya hemos dicho más arriba, una fórmula atómica sólo puede evaluarse si sus variables toman valores concretos. Por ejemplo, si $H(x)$ significa " x es hombre", la fórmula será verdadera para $x = \text{Sócrates}$, pero falsa para $x = \text{Rocinante}$, y, en general, no podremos decir que la fórmula atómica $H(x)$ sea verdadera ni falsa.

De igual modo, una sentencia abierta no puede evaluarse si no se cierra. Pero hemos de considerar varias posibilidades:

a) Si una sentencia es cerrada porque a todas las variables que intervienen en ella se les ha asignado un valor constante, entonces las fórmulas atómicas juegan el mismo papel que las variables proposicionales, y las distintas evaluaciones de la sentencia pueden escribirse en una tabla de verdad, con una línea para cada una de las posibles evaluaciones del conjunto de fórmulas atómicas.

b) Si la sentencia es cerrada porque todas sus variables están cuantificadas, entonces cabe considerar dos casos:

b1) Si el universo del discurso (conjunto en el que toman valores las variables) es finito entonces el cuantificador universal puede sustituirse por un número finito de conjunciones, y el cuantificador existencial, por un número finito de disyunciones. En efecto, si, por ejemplo, los valores posibles de x son $\{a, b, c, d\}$, la sentencia cerrada

$$(\forall x) (P(x))$$

es equivalente a

$$P(a) \wedge P(b) \wedge P(c) \wedge P(d)$$

y la sentencia cerrada

$$(\exists x) (P(x))$$

es equivalente a

$$P(a) \vee P(b) \vee P(c) \vee P(d)$$

Estas equivalencias proceden de las mismas definiciones de los cuantificadores, que, realmente, son generalizaciones de las operaciones de

conjunción y disyunción para sentencias construidas con variables de cualquier rango. En este caso de universo finito, podemos escribir la tabla de verdad de cualquier sentencia en la que intervenga $P(x)$ con x cuantificada, considerando todas las evaluaciones posibles del conjunto $\{P(a), P(b), P(c), P(d)\}$ (que a todos los efectos es lo mismo que un conjunto de variables proposicionales).

- b2) Lo más normal es que el universo del discurso sea infinito (o inabordable). En tal caso, es claro que no podemos construir ninguna tabla de verdad. Pese a ello, hay sentencias que siempre son verdaderas. Por ejemplo, nadie dudará de que

$$(\forall x) (P(x)) \rightarrow (\exists x) (P(x))$$

es siempre verdadera, sea finito o infinito el universo del discurso.

A las sentencias que son siempre verdaderas se les llama, en lógica de proposiciones, tautologías. En lógica de predicados se les llama *sentencias válidas* (una tautología es un caso particular de sentencia válida en la que sólo figuran constantes).

1.7. Ejemplos de formalización e interpretación

Ejemplo 1.7.1. Consideremos el razonamiento con el que empezábamos en el capítulo 1:

Premisa 1: Todos los libros sobre informática son terriblemente aburridos.

Premisa 2: Éste es un libro sobre informática.

Conclusión: Este libro es terriblemente aburrido.

Creando los símbolos I, A, e , con la interpretación $i(I)$ = 'libro de informática', $i(A)$ = 'libro terriblemente aburrido', $i(e)$ = 'este libro', la formalización es:

$$\text{Premisa1: } (\forall x) (I(x) \rightarrow A(x))$$

$$\text{Premisa2: } I(e)$$

$$\text{Conclusión : } A(e)$$

Si el razonamiento es correcto, la sentencia

$$[(\forall x) (I(x) \rightarrow A(x))] \wedge I(e) \rightarrow A(e)$$

tiene que ser válida, o, lo que es lo mismo, todas las interpretaciones que satisfacen a la conjunción de las dos premisas han de satisfacer también a la

conclusión para que el razonamiento sea válido. Ahora bien, como el universo del discurso es inabordable, no podemos comprobar tal cosa construyendo una tabla de verdad. Pero hagámoslo considerando que el universo se restringiese a tres libros, éste (*e*) y otros dos (*o1* y *o2*). La premisa *P1* se escribiría entonces como $P1e \wedge P1o1 \wedge P1o2$, con

$$P1e: I(e) \rightarrow A(e)$$

$$P1o1: I(o1) \rightarrow A(o1)$$

$$P1o2: I(o2) \rightarrow A(o2)$$

Y la tabla de verdad sería la siguiente:

<i>I(e)</i>	<i>A(e)</i>	<i>I(o1)</i>	<i>A(o1)</i>	<i>I(o2)</i>	<i>A(o2)</i>	<i>P1e</i>	<i>P1o1</i>	<i>P1o2</i>	<i>P1</i>	$P2 \equiv I(e)$	$C \equiv A(e)$	$P1 \wedge P2 \rightarrow C$
0	0	0	0	0	0	1	1	1	1	0	0	1
0	0	0	0	0	1	1	1	1	1	0	0	1
0	0	0	0	1	0	1	1	0	0	0	0	1
0	0	0	0	1	1	1	1	1	1	0	0	1
0	0	1	0	0	0	1	1	1	1	0	0	1
0	0	0	1	0	1	1	1	1	1	0	0	1
0	0	0	1	1	0	1	1	0	0	0	0	1
0	0	0	1	1	1	1	1	1	1	0	0	1
0	0	1	0	0	0	1	0	1	0	0	0	1
0	0	1	0	0	1	1	0	1	0	0	0	1
0	0	1	0	1	0	1	0	0	0	0	0	1
0	0	1	0	1	1	1	0	1	0	0	0	1
0	0	1	1	0	0	1	1	1	1	0	0	1
0	0	1	1	0	1	1	1	1	1	0	0	1
0	0	1	1	1	0	1	1	0	0	0	0	1
0	0	1	1	1	1	1	1	1	1	0	0	1
0	1	0	0	0	0	1	1	1	1	0	1	1
0	1	0	0	0	1	1	1	1	1	0	1	1
0	1	0	0	1	0	1	1	0	0	0	1	1

$I(e)$	$A(e)$	$I(01)$	$A(01)$	$I(02)$	$A(02)$	$P1e$	$P1o1$	$P1o2$	$P1$	$P2 \equiv I(e)$	$C \equiv A(e)$	$P1 \wedge P2 \rightarrow C$
0	1	0	0	1	1	1	1	1	1	0	1	1
0	1	0	1	0	0	1	1	1	1	0	1	1
0	1	0	1	0	1	1	1	1	1	0	1	1
0	1	0	1	1	0	1	1	0	0	0	1	1
0	1	0	1	1	1	1	1	1	0	0	1	1
0	1	1	0	0	0	1	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0	1	1
0	1	1	0	1	0	1	0	0	0	0	1	1
0	1	1	0	1	1	1	0	1	0	0	1	1
0	1	1	1	0	0	1	1	1	1	0	1	1
0	1	1	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	0	1	1	0	0	0	1	1
0	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	0	0	0	1	1	0	1	0	1
1	0	0	0	0	1	0	1	1	0	1	0	1
1	0	0	0	1	0	0	1	0	0	1	0	1
1	0	0	0	1	1	0	1	1	0	1	0	1
1	0	0	1	0	0	0	1	1	0	1	0	1
1	0	0	1	0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	1	0	0	1	0	1
1	0	0	1	1	1	0	1	1	0	1	0	1
1	0	1	0	0	0	0	0	1	0	1	0	1
1	0	1	0	0	1	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0	0	0	1	0	1
1	0	1	0	1	1	0	0	1	0	1	0	1
1	0	1	1	0	0	0	1	1	0	1	0	1
1	0	1	1	0	1	0	1	1	0	1	0	1

$I(e)$	$A(e)$	$I(01)$	$A(01)$	$I(02)$	$A(02)$	$P1e$	$P1o1$	$P1o2$	$P1$	$P2 \equiv I(e)$	$C \equiv A(e)$	$P1 \wedge P2 \rightarrow C$
1	0	1	1	1	0	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1	1	0	1	0	1
1	1	0	0	0	0	1	1	1	1	1	1	1
1	1	0	0	0	1	1	1	1	1	1	1	1
1	1	0	0	1	0	1	1	0	0	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1
1	1	0	1	0	0	1	1	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1	0	0	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	0	0	0	1	0	1	0	1	1	1
1	1	1	0	0	1	1	0	1	0	1	1	1
1	1	1	0	1	0	1	0	0	0	1	1	1
1	1	1	0	1	1	1	0	1	0	1	1	1
1	1	1	1	0	0	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1

Tras escribir toda esta tabla, comprobamos que, para este universo finito, la sentencia que formaliza al razonamiento es una tautología, y, por tanto, el razonamiento es válido. Pero comprobamos también que, además de tedioso, este procedimiento (que en lógica de proposiciones podría tener algún sentido) es inútil, porque, en principio, no nos garantiza que el razonamiento siga siendo válido para un universo del discurso mayor.

Se ha demostrado que, si hay n predicados monádicos, cualquier sentencia válida en un universo que tenga 2^n elementos es válida en todo universo. En este ejemplo, como hay dos predicados monádicos, bastaría comprobar la validez en un universo de cuatro elementos, lo cual sería factible. Pero para un número mayor de predicados el procedimiento deja de ser práctico, y si hay predicados poliádicos no nos garantiza nada. De hecho, también se ha demostrado que no hay un procedimiento general que permita determinar la validez de cualquier sentencia

del cálculo de predicados. Por ello, se dice que el cálculo de predicados es *indecidable*. Sin embargo, sí existe un procedimiento tal que si una sentencia es válida termina dictaminándolo (y si no lo es, no termina), por lo que también se dice que el cálculo de predicados es *semidecidible*.

En cualquier caso, lo que nos interesa a efectos prácticos de aplicación de la lógica a la informática es disponer de sistemas de inferencia. En el apartado 4.7 generalizaremos el método de resolución que habíamos visto para la lógica de proposiciones. En los ejemplos que siguen nos limitaremos a formalizar los razonamientos, dejando al lector la construcción de tablas de verdad para comprobar su validez con universos finitos.

Ejemplo 1.7.2.

ningún profesional es diletante
todos los buenos informáticos son profesionales
ningún buen informático es diletante

Con los predicados P ($i(P)$ = profesional), D ($i(D)$ = diletante) e I ($i(I)$ = buen informático), una formalización de este razonamiento (que es un caso del llamado "modo silogístico Celarent" en la lógica clásica) es:

$$\begin{array}{l} P1: (\forall x) (P(x) \rightarrow \neg D(x)) \\ P2: (\forall x) (I(x) \rightarrow P(x)) \\ \hline C: (\forall x) (I(x) \rightarrow \neg D(x)) \end{array}$$

Y la sentencia global sería: $P1 \wedge P2 \rightarrow C$.

Teniendo en cuenta las relaciones que hay entre las conectivas y entre los cuantificadores, la formalización no es única. Por ejemplo, la primera premisa puede también escribirse así:

$$P1: \neg(\exists x) (P(x) \wedge D(x))$$

Ejemplo 1.7.3. Definiendo los predicados J , C , D y S en sustitución de las variables proposicionales j , c , d y s , el ejemplo 1.4.4 del capítulo 2 se formalizaría así:

$$\begin{array}{l} P1: \neg(\exists x) (J(x) \wedge C(x)) \\ P2: (\forall x) (\neg J(x) \rightarrow \neg D(x)) \\ P3: (\forall x) (S(x) \rightarrow C(x)) \\ \hline C: (\forall x) (S(x) \rightarrow C(x)) \end{array}$$

$P1$ puede también escribirse de esta otra forma equivalente:

$$(\forall x) (J(x) \rightarrow \neg C(x))$$

Ejemplo 1.7.4.

Ana es madre de Luis

José es padre de Ana

un abuelo de una persona es alguien que es padre del padre o de la madre de esa persona

José es abuelo de Luis

Definiendo los predicados diádicos $P(x,y)$ (x es padre de y), $M(x,y)$ (x es madre de y) y $A(x,y)$ (x es abuelo de y), podemos establecer la siguiente formalización:

$P1: M(a,l)$

$P2: P(j,a)$

$P3: (\forall x) (\forall y) (A(x,y) \rightarrow (P(x,y) \vee M(x,y)))$

$C: A(j,l)$

Obsérvese que la premisa 3 establece una *definición de la relación* "abuelo de" en función de las relaciones "padre de" y "madre de". Las premisas 1 y 2 son *hechos* concretos que junto con esa definición nos permiten inferir otro hecho.

Ejemplo 1.7.5.

"Un antepasado de una persona es alguien que o bien es padre o madre de esa persona o bien es antepasado de su padre o de su madre".

Esta es una *definición recursiva* de la relación "antepasado de". Dados unos hechos (relaciones "padre de" y "madre de" definidas sobre unos individuos concretos), se podrán obtener conclusiones sobre las relaciones "antepasado de" entre esos individuos. En lógica de predicados, podemos formalizar la definición con la siguiente sentencia (ahora, con $T(x,y)$ representamos " x es antepasado de y ", y, para no hacer la sentencia excesivamente larga, con $P(x,y)$ representamos " x es padre o madre de y "; si quisiéramos conservar las dos relaciones diferenciadas, tendríamos que escribir $(P(x,y) \vee M(x,y))$ en donde aparece $P(x,y)$):

$$(\forall x) (\forall y) (\forall z) [(P(x,y) \vee (T(x,z) \wedge P(z,y))) \rightarrow T(x,y)]$$

Pero, para mayor claridad, es preferible descomponer la sentencia en dos, una no recursiva y la otra sí:

$$(\forall x) (\forall y) (P(x,y) \rightarrow T(x,y))$$

$$(\forall x) (\forall y) (\forall z) ((T(x,z) \wedge P(z,y)) \rightarrow T(x,y))$$

Esta descomposición es posible porque la sentencia $P1 \vee P2 \rightarrow C$ es equivalente a $(P1 \rightarrow C) \wedge (P2 \rightarrow C)$.

1.8. Funciones

Para terminar esta introducción informal, tenemos que hacer referencia a las *funciones*. Se trata de una construcción del cálculo de predicados que no se utilizaba en la lógica clásica, pero que, como veremos (en el apartado 4.5), es necesaria para poder expresar las sentencias en forma clausulada como paso previo a la aplicación de la resolución.

Las funciones, como los predicados, tienen argumentos que pueden ser constantes o variables. Pero, a diferencia de los predicados, no representan ninguna propiedad o relación entre los argumentos que pueda evaluarse como verdadera o falsa. Una función en cálculo de predicados es una función matemática definida en el universo del discurso. Por ejemplo, podemos definir la función "padre", p . Aplicada a un individuo, nos daría como resultado otro individuo, $p(x)$ (su padre). Si definimos un predicado de igualdad, $I(x,y)$, que es verdadero cuando x es el mismo individuo que y , y falso en otro caso, sería lo mismo escribir $P(x,y)$ (" x es padre de y ") que $I(x,p(y))$ ("el individuo x es el mismo individuo que es padre de y "). Vemos con este ejemplo que los argumentos de los predicados (y de las mismas funciones) pueden ser funciones, lo cual nos lleva a ampliar el concepto de fórmula atómica, cosa que haremos enseguida.

2. Sintaxis

2.1. Alfabeto

Utilizaremos los siguientes símbolos (en su caso, y si es menester, con subíndices):

- Variables proposicionales: p, q, r, \dots (normalmente, no serán necesarias; en su lugar tendremos fórmulas atómicas).
- Variables: x, y, z, u, v, w
- Constantes: a, b, c, \dots
- Símbolos de función: f, g, h
- Símbolos de predicado: P, Q, R
- Conectivas (las mismas de la lógica de proposiciones)
- Cuantificadores: \forall, \exists
- Símbolos de puntuación: $(,), [,], \{, \}$
- Metasímbolos:

A, B, C, \dots para cualquier sentencia.

$A(x)$ para una sentencia con la variable libre x .

k para cualquier conectiva binaria.

L para cualquier literal (fórmula atómica positiva o negativa).

t para cualquier "término", una construcción que enseguida vamos a definir.

Igual que hacíamos en lógica de proposiciones, podemos definir una *expresión* como una secuencia cualquiera de símbolos del alfabeto, y una *sentencia* como una expresión "bien construida" de acuerdo con las reglas propias del lenguaje y que vamos a ver a continuación.

2.2. Términos y fórmulas atómicas

Definición 2.2.1. Un *término* se define, recursivamente, así:

- las constantes y las variables son términos
- si f es un símbolo de función y t_1, t_2, \dots, t_n son términos, entonces $f(t_1, t_2, \dots, t_n)$ es un término.

(Podríamos haber dado una definición equivalente, no recursiva, definiendo previamente una "secuencia de formación de términos", lo mismo que hacíamos en lógica de proposiciones para las sentencias).

Definición 2.2.2. Una fórmula atómica es una expresión de la forma $P(t_1, t_2, \dots, t_n)$, donde P es un símbolo de predicado y t_1, t_2, \dots, t_n son términos.

2.3. Sentencias

Definición 2.3.1. Llamaremos *secuencia de formación* a toda secuencia finita de expresiones, A_1, A_2, \dots, A_n en la que cada A_i satisface al menos una de las cinco condiciones siguientes:

- A_i es una fórmula atómica (o una variable proposicional);
- existe un j menor que i tal que A_i es el resultado de anteponer el símbolo " \neg " a A_j ;
- existen j y h menores que i tales que A_i es el resultado de enlazar A_j y A_h con k ;

- d) existe un j menor que i tal que A_i es lo mismo que $(\forall x) (A_j(x))$, donde x es cualquier variable;
- e) existe un j menor que i tal que A_i es lo mismo que $(\exists x) (A_j(x))$, donde x es cualquier variable.

Definición 2.3.2. Llamaremos *sentencia* (o *fórmula molecular*) a toda expresión A_n tal que existe una secuencia de formación A_1, A_2, \dots, A_n . (Todas las expresiones de la secuencia serán sentencias).

Alternativamente, podemos definir la sentencia de modo recursivo, mediante cinco reglas de formación:

RF1: Una fórmula atómica (o una variable proposicional) es una sentencia.

RF2: Si A es una sentencia, $\neg A$ también lo es.

RF3: Si A y B son sentencias, $A \wedge B$ también lo es.

RF4: Si $A(x)$ es una sentencia con la variable x , $(\forall x) (A(x))$ también lo es.

RF5: Si $A(x)$ es una sentencia con la variable x , $(\exists x) (A(x))$ también lo es.

Según cualquiera de estas definiciones, una expresión como

$$(\forall x) (\exists y) (P(x,y))$$

no sería una sentencia. En su lugar habría que escribir

$$(\forall x) ((\exists y) P(x,y))$$

No obstante, por comodidad, utilizaremos preferentemente la primera notación.

2.4. Sentencias abiertas y cerradas

Definición 2.4.1. Llamaremos *alcance* de un cuantificador a la sentencia A que le sigue en las reglas de formación RF4 y RF5. Diremos que la variable que sigue al símbolo de cuantificación está *cuantificada* en la sentencia A . Una variable no cuantificada se dice que está *libre*. En las condiciones d) y e) y en las reglas RF3 y RF4 del apartado anterior debe entenderse que la variable x en $A(x)$ está libre.

Definición 2.4.2. Diremos que una sentencia A es *cerrada* si todas las variables que intervienen en las fórmulas atómicas que la componen están cuantificadas. En caso contrario, diremos que la sentencia A es *abierta*.

Aunque aún no hemos entrado en el campo de la semántica, por lo visto en el apartado anterior sabemos que sólo a las sentencias cerradas se les puede dar una evaluación ("verdadera" o "falsa", en el caso binario). En lo sucesivo, salvo que digamos lo contrario, supondremos que todas las sentencias son cerradas. De hecho, en otros libros, el nombre de "sentencia" se reserva para lo que aquí llamamos "sentencias cerradas", y lo que llamamos "sentencia" se denomina "fórmula bien formada" (en los libros escritos en inglés, "wff", de "well formed formula"), denominación, cuando menos, discutible: ¿qué sería una "fórmula mal formada"?; con nuestra terminología, una expresión que no es sentencia, pero ¿sería entonces lícito llamarle "fórmula"?

2.5. Axiomas, demostraciones y teoremas

2.5.1. Axiomas

Los conceptos de sistema axiomático, de completitud y de consistencia ya se estudiaron de una manera general en el capítulo 2 (apartado 1.5). El sistema axiomático de la lógica de predicados es una extensión del visto para la lógica de proposiciones (o éste es una restricción del primero); por ello, aquí nos iremos refiriendo continuamente al apartado 2.5 del capítulo 2. Y, en particular, los axiomas son los mismos presentados en 2.5.1, a los que ahora añadimos estos dos:

$$A5: (\forall x) (P(x)) \rightarrow P(a)$$

$$A6: (\forall x) (p \rightarrow P(x)) \rightarrow (p \rightarrow (\forall x) (P(x)))$$

Las leyes que interrelacionan a los cuantificadores,

$$(\forall x) (P(x)) \leftrightarrow \neg(\exists x) (\neg P(x))$$

$$(\exists x) (P(x)) \leftrightarrow \neg(\forall x) (\neg P(x))$$

pueden considerarse como definición de un símbolo en función de otro, o, si se prefiere que ambos símbolos sean primitivos, una sería un axioma y la otra un teorema.

El axioma A5 (llamado "*ley de especificación*") significa que si algo (P) puede afirmarse de todos los individuos, entonces puede afirmarse de uno cualquiera de ellos: a es una constante arbitraria definida sobre el mismo universo del discurso que x . Igualmente, en A6 p es una variable proposicional arbitraria.

2.5.2. Sustitución

En lógica de proposiciones, la sustitución era de variables proposicionales por sentencias. Ahora, además, podremos sustituir fórmulas atómicas por sentencias en las que figuren los mismos argumentos, y variables por términos en los que no aparezcan otras variables de la sentencia.

Definición 2.5.2.1. Dadas unas variables proposicionales p_i , unas sentencias cerradas B_i , unas fórmulas atómicas $F_j = P_j(x_{j1}, \dots, x_{jn}, a_1, \dots, a_m)$, unas sentencias S_j con las variables libres $x_{j1} \dots x_{jn}$ y las constantes a_1, \dots, a_m , unas variables x_k y unos términos t_k , llamaremos *sustitución* a un conjunto de pares ordenados:

$$s = \{ \dots, B_i/p_i, \dots, S_j/F_j, \dots, t_k/x_k, \dots \}$$

Definición 2.5.2.2. Dadas una sentencia A que contiene las variables proposicionales p_i , las fórmulas atómicas F_j y las variables x_k , y una sustitución s , la *operación de sustitución* de unas por otras en A consiste en poner en A :

- a) en todos los lugares donde aparezca p_i , B_i ,
- b) en todos los lugares donde aparezca F_j , S_j ,
- c) en todos los lugares donde aparezca x_k , t_k , salvo detrás de los cuantificadores, donde se pondrán las variables que aparecen en t_k precedidas del mismo cuantificador que afectaba a x_k . Además, para que la regla de sustitución que veremos luego sea correcta, t_k no puede contener variables que ya figuren en A .

La justificación de esta última restricción puede verse claramente con un sencillo ejemplo: consideremos la sentencia

$$(\forall x) (\exists y) (D(x, y))$$

donde $D(x, y)$ significa " x es diferente de y "; podemos hacer la sustitución $s = \{z/x\}$, con la que obtenemos

$$(\forall z) (\exists y) (D(z, y))$$

pero no $s = \{y/x\}$, con la que resultaría

$$(\forall y) (\exists y) (D(y, y))$$

Teorema 2.5.2.3. Si A es una sentencia y s una sustitución, el resultado de efectuar la operación de sustitución de s en A es otra sentencia que representaremos como As . La demostración, que omitimos, es sencilla: basta comprobar que As tiene una secuencia de formación.

2.5.3. Demostraciones y teoremas

Para este apartado es íntegramente aplicable todo lo dicho en los apartados 2.5.3 y 2.5.4 del capítulo 2.

2.5.4. Ejemplos de demostraciones

Todos los teoremas de la lógica de proposiciones son aplicables también en lógica de predicados. Veamos algunos otros específicos de ésta.

Teorema 1: $P(a) \rightarrow (\exists x) (P(x))$ ("Ley de inespecificación")

Demostración:

1. $(\forall x) (\neg P(x)) \rightarrow \neg P(a)$
(Por sustitución en A5: $\{\neg P/P\}$)
2. $(p \rightarrow \neg q) \rightarrow (q \rightarrow \neg p)$
(Teorema de la lógica de proposiciones)
3. $((\forall x) (\neg P(x)) \rightarrow \neg P(a)) \rightarrow (P(a) \rightarrow \neg(\forall x) (\neg P(x)))$
(Por sustitución en 2: $\{(\forall x) (\neg P(x))/p, P(a)/q\}$).
4. $P(a) \rightarrow \neg(\forall x) (\neg P(x))$
(Por separación de 1 y 3)
5. $P(a) \rightarrow (\exists x) (P(x))$
(Por definición de " \exists ")

Teorema 2: $(\forall x) (P(x)) \rightarrow (\exists x) (P(x))$

Demostración:

1. $((\forall x) P(x) \rightarrow P(a)) \wedge (P(a) \rightarrow (\exists x) (P(x)))$
(Por unión de A5 y el teorema 1)
2. $((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$
(Ley de transitividad de la lógica de proposiciones)
3. $((\forall x) P(x) \rightarrow P(a)) \wedge (P(a) \rightarrow (\exists x) (P(x))) \rightarrow ((\forall x) (P(x)) \rightarrow (\exists x) (P(x)))$
(Por sustitución en 2: $\{(\forall x) (P(x))/p, P(a)/q, (\exists x) (P(x))/r\}$)
4. $(\forall x) (P(x)) \rightarrow (\exists x) (P(x))$
(Por separación de 1 y 3)

2.5.5. Algunos teoremas útiles

La mayoría de las leyes de la lógica de proposiciones se pueden generalizar a la lógica de predicados. Por ejemplo:

- *Ley del tercio excluso:*

$$(\forall x) (P(x) \vee \neg P(x))$$

- *Ley de distribución del cuantificador universal sobre la conjunción:*

$$(\forall x) (P(x) \wedge Q(x)) \rightarrow [(\forall x) (P(x)) \wedge (\forall y) (P(y))]$$

- *Leyes de modus ponens y modus tollens:*

$$((\forall x) (P(x) \rightarrow Q(x)) \wedge P(a)) \rightarrow Q(a)$$

$$((\forall x) (P(x) \rightarrow Q(x)) \wedge \neg Q(a)) \rightarrow \neg P(a)$$

- *Leyes de inferencia de la alternativa:*

$$(\neg P(a) \wedge (\forall x) (P(x) \vee Q(x))) \rightarrow Q(a)$$

$$(P(a) \wedge (\forall x) (\neg P(x) \vee \neg Q(x))) \rightarrow \neg Q(a)$$

Todas estas leyes, escritas para predicados monádicos, pueden generalizarse para predicados poliádicos. Por ejemplo, la de modus ponens para diádicos es:

$$((\forall x) (\forall y) (P(x, y) \rightarrow Q(x, y)) \wedge P(a, b)) \rightarrow Q(a, b)$$

Para la regla de resolución (que veremos en el apartado 4.7) utilizaremos una generalización de las leyes de inferencia de la alternativa:

$$(\forall x_1) \dots (\forall x_n) ((\neg P(x_1, \dots, x_n) \vee A) \wedge (P(x_1, \dots, x_n) \vee B)) \rightarrow A \vee B$$

donde A y B son sentencias cualesquiera.

3. Semántica

Todo lo explicado en el apartado 3 del capítulo 2 es extensible a la lógica de predicados. Vamos a hacerlo, recomendando al lector que alterne el estudio de lo que sigue con la revisión de lo que allí decíamos. Empezaremos con la definición de *conceptualización*.

Definición 3.1. Una *conceptualización* es un modelo conceptual que consta de:

- Un *universo del discurso*, \mathcal{U} , conjunto (que puede ser infinito) de individuos u objetos a considerar.

- Un conjunto finito (que puede ser vacío), \mathbf{R} , de *relaciones* entre los elementos de \mathbf{U} .
- Un conjunto finito (que puede ser vacío), \mathbf{F} , de *funciones* que hacen corresponder a ciertos elementos de \mathbf{U} otros elementos de \mathbf{U} .

Una *relación de grado n* es un subconjunto del producto cartesiano \mathbf{U}^n , es decir, es un conjunto de tuplas $\{ (e_{i1}, e_{i2}, \dots, e_{in}) \}$, donde $e_{ij} \in \mathbf{U}$. Una *propiedad* es una relación de grado 1.

Una *función de grado n* es una aplicación $\mathbf{U}^n \rightarrow \mathbf{U}$, es decir, hace corresponder a cada combinación de n elementos de \mathbf{U} otro elemento de \mathbf{U} .

3.1. Interpretaciones, asignaciones y evaluaciones

Definición 3.1.1. Una *interpretación*, i , es una función que aplica elementos del lenguaje en elementos de la conceptualización, y que debe satisfacer las siguientes condiciones:

- Si c es un símbolo de constante, entonces $i(c) \in \mathbf{U}$
(es decir, las constantes *representan* individuos del universo del discurso).
- Si P es un símbolo de predicado de grado n (es decir, que se aplica a n términos para formar una fórmula atómica), entonces $i(P) \subset \mathbf{U}^n$ (es decir, los símbolos de predicado *representan* relaciones de la conceptualización).
- Si f es un símbolo de función de grado n , entonces $i(f) = \mathbf{U}^n \rightarrow \mathbf{U}$ (es decir, los símbolos de función *representan* funciones de la conceptualización).

El formalismo matemático puede ofuscar al lector, cuando en realidad el concepto de interpretación es simple. Se trata sencillamente de formalizar la noción de que los símbolos *representan* aspectos de la realidad modelada en la conceptualización. Como ejemplo, volvamos a las relaciones familiares aludidas más arriba, en la presentación informal del apartado 1 (ejemplos 1.7.4 y 1.7.5). Hacer una conceptualización significa *acotar la realidad*, es decir, en este caso, listar todos los individuos que consideramos y todas las relaciones entre ellos. Consideremos la siguiente conceptualización:

$\mathbf{U} = \{\text{José, Ana, Luis, Angel, Baltasar, Bartolomé, Belén, Carmen, Diego}\}$

$\mathbf{R} = \{\text{Padre, Madre, Antepasado}\}$, relaciones de grado 2 definidas por:

Padre = $\{ (\text{José, Ana}), (\text{José, Baltasar}), (\text{Baltasar, Diego}),$
 $(\text{Bartolomé, Angel}), (\text{Bartolomé, Belén}) \}$

Madre={ (Ana, Luis), (Ana, Bartolomé), (Ana, Carmen) }
 (Es decir, José es padre de Ana, etc.)

Antepasado=Padre \cup Madre \cup {(José, Luis), (José, Bartolomé),
 (José, Carmen), (José, Angel),
 (José, Belén), (Ana, Angel),
 (Ana, Belén), (José, Diego)}

(Es decir, todas las parejas que están en la relación "Padre" y las que están en la relación "Madre" están también en la relación "Antepasado", y, además, José es antepasado de Luis, etc.).

Para formalizar las sentencias, creamos los símbolos j , $a1$, $a2$, etc. para las constantes, y P , M y T para las relaciones, con la siguiente función de interpretación:

$i(j)$ =José; $i(a1)$ =Ana; $i(l)$ =Luis; $i(a2)$ =Angel;
 $i(b1)$ =Baltasar; $i(b2)$ =Bartolomé; $i(b3)$ =Belén;
 $i(i)$ =Carmen; $i(d)$ =Diego

 $i(P)$ =Padre; $i(M)$ =Madre
 $i(T)$ =Antepasado

Esta sería la *interpretación pretendida*, o "natural", la que sugieren los símbolos " P ", " M " y " T ". Pero en teoría podemos hacer cualquier otra interpretación. Por ejemplo, con una interpretación i' igual a i para las constantes pero tal que:

$i'(P)$ ={ (Ana, José), (Baltasar, José), (Diego, Baltasar),
 (Angel, Bartolomé), (Belén, Bartolomé) }

estaríamos interpretando (supuesto que la realidad no ha cambiado, o sea, que la relación "Padre" es la misma) que el símbolo de predicado " P " significa en realidad "*hijo/a*".

Definición 3.1.2. Una *asignación de variables*, A , es una función que hace corresponder elementos de \mathcal{U} a las variables que figuran en las sentencias. Las constantes corresponden a los elementos de \mathcal{U} según define la interpretación, y las funciones se aplican también en elementos de \mathcal{U} , según su interpretación. En general, dada una interpretación i y una asignación de variables A , podemos extender A a una *asignación de términos*:

- Si c es un símbolo de constante, $A(c) = i(c)$;
- Si f es un símbolo de función,

$$A(f(t_1, t_2, \dots, t_n)) = F(x_1, x_2, \dots, x_n), \text{ donde } F = i(f) \text{ y } x_i = A(t_i)$$

Para ilustrarlo con un ejemplo, supongamos que a la conceptualización anterior añadimos la función de grado 1 "FPadre", que aplicada a Aña da como resultado José, aplicada a Baltasar da como resultado José, etc. ("FPadre" contiene la misma información que la relación "Padre" pero expresada como función). Representemos esta función con el símbolo " p ", es decir, $i(p)$ =FPadre y consideremos el término $t = p(p(x))$. Dada, por ejemplo, la asignación de variables x =Diego, resulta:

$$A(t) = A(p(p(x))) = \text{FPadre}(A(p(x))) = \text{FPadre}(\text{FPadre}(A(x))) = \text{FPadre}(\text{FPadre}(\text{Diego})) = \text{FPadre}(\text{Baltasar}) = \text{José}$$

3.2. Satisfacción

Como en el capítulo 2, podemos definir en general un conjunto de valores de verdad (definición 3.1.2), evaluar en él las fórmulas atómicas (definición 3.1.3) y extender estas evaluaciones a las sentencias (teorema 3.1.4). Pero ahora hay que tener en cuenta que las fórmulas atómicas (que sustituyen a las variables proposicionales) contienen normalmente variables, y solamente se puede evaluar una fórmula atómica si sus variables están asignadas. Una evaluación es, pues, relativa no sólo a la interpretación, sino también a la asignación de variables. En adelante abreviaremos con " iA " la expresión "interpretación y asignación de variables".

En el resto de este capítulo nos limitaremos a considerar evaluaciones binarias (definición 3.2.1 del capítulo 2).

Una misma sentencia puede ser verdadera o falsa en una determinada conceptualización dependiendo de la interpretación y de la asignación de variables. Así, en la conceptualización de los Padres y Antepasados, la sentencia " $P(x,y)$ " es verdadera para la interpretación i (la que llamábamos interpretación "pretendida" o "natural") y las asignaciones $A_1(x)$ =José, $A_1(y)$ =Ana; $A_2(x)$ =José, $A_2(y)$ =Baltasar, etc., pero es falsa para las mismas asignaciones si adoptamos la interpretación i' , y también es falsa para la interpretación i pero con otras asignaciones.

Definición 3.2.1. En una conceptualización, diremos que una interpretación conjuntamente con una asignación de variables, iA , *satisface* a una sentencia si la sentencia es verdadera para esa iA . También diremos que la sentencia *se satisface* para esa iA . La notación¹ para expresar que la sentencia S se satisface para iA es: $\models_{iA} S$

¹ " S " es aquí un metasímbolo para representar a una sentencia cualquiera. Más adelante utilizaremos también " P " y " C " como metasímbolos para representar sentencias que son premisas y conclusiones de razonamientos. Por otra parte, obsérvese que " \models_{iA} ", como otros símbolos parecidos que iremos introduciendo, son predicados de orden superior, ya que representan propiedades de sentencias del cálculo de predicados de primer orden.

Recordemos que las sentencias se forman siguiendo las reglas sintácticas explicadas en el apartado 2.3. Si para cada una de las reglas (RF1 a RF5) establecemos las condiciones bajo las cuales la sentencia se satisface, estas condiciones definen la *semántica de la lógica de predicados de primer orden*. Son las siguientes:

- Si la sentencia es una fórmula atómica, $P(t_1, t_2, \dots, t_n)$, entonces

$$\models_{iA} P(t_1, t_2, \dots, t_n) \text{ sii } (A(t_1), A(t_2), \dots, A(t_n)) \in i(P)$$

(Una fórmula atómica se satisface si y sólo si las asignaciones de sus argumentos resultan en una tupla incluida en la interpretación del predicado).

- Si la sentencia es la negación de otra, entonces

$$\models_{iA} (\neg S) \text{ sii no es el caso que } \models_{iA} S$$

(La negación de una sentencia se satisface si y sólo si no se satisface la sentencia).

- Si la sentencia es la conjunción de otras dos, entonces

$$\models_{iA} (S1 \wedge S2) \text{ sii } \models_{iA} S1 \text{ y } \models_{iA} S2$$

(La conjunción de dos sentencias se satisface si y sólo si se satisfacen ambas. Esta condición puede generalizarse a la conjunción de un número cualquiera de sentencias).

- Si la sentencia es la disyunción de otras dos, entonces

$$\models_{iA} (S1 \vee S2) \text{ sii o bien } \models_{iA} S1 \text{ o bien } \models_{iA} S2 \text{ o ambos}$$

(La disyunción de dos sentencias se satisface si y sólo si se satisface una cualquiera de ellas o ambas, o sea, es una disyunción "incluyente" (no excluyente). Esta condición puede generalizarse a la disyunción de un número cualquiera de sentencias).

- Si la sentencia es un condicional, entonces

$$\models_{iA} (S1 \rightarrow S2) \text{ sii no es el caso que } (\models_{iA} S1 \text{ y no } \models_{iA} S2)$$

(Una sentencia condicional se satisface siempre que no ocurra que su parte izquierda, o *antecedente*, se satisface y su parte derecha, o *consecuente*, no se satisface).

- Si la sentencia es un bicondicional, entonces

$$\models_{iA} (S1 \leftrightarrow S2) \text{ sii } \models_{iA} (S1 \rightarrow S2) \text{ y } \models_{iA} (S2 \rightarrow S1)$$

(Una sentencia bicondicional se satisface si y sólo si se satisfacen los dos condicionales en que puede descomponerse).

- Si la sentencia se forma cuantificando universalmente a una variable de otra sentencia, entonces

$$\models_{iA} (\forall x) (S) \text{ sii para todo } c \in \mathcal{U}, \models_{iA} (S), \\ \text{con } A'(x) = c, A'(y) = A(y) \text{ para } y \neq x$$

(Una sentencia cuantificada universalmente en una variable se satisface si y sólo si la sentencia que está dentro del alcance del cuantificador se satisface para todas las asignaciones de la variable cuantificada).

- Si la sentencia se forma cuantificando existencialmente a una variable de otra sentencia, entonces

$$\models_{iA} (\exists x) (S) \text{ sii para algún } c \in \mathcal{U}, \models_{iA} (S), \\ \text{con } A'(x) = c, A'(y) = A(y) \text{ para } y \neq x$$

(Una sentencia cuantificada existencialmente se satisface si y sólo si la sentencia que está dentro del alcance del cuantificador se satisface para una o más de las asignaciones de la variable cuantificada).

El concepto de satisfacción define la noción *relativa* de verdad o falsedad: en general, una misma sentencia puede ser verdadera o falsa, dependiendo de la interpretación y de la asignación de variables (*iA*). Naturalmente, en el caso de que haya una variable cuantificada, para que la sentencia se satisfaga para una *iA* debe hacerlo para la asignación *A* de todas las variables no cuantificadas y para todas las asignaciones posibles de la variable cuantificada (en el caso de cuantificación universal) o para alguna de ellas (en el caso de cuantificación existencial).

Recordemos (apartado 1.3 del capítulo 2) que la satisfacción del condicional refleja el hecho de que éste se entiende como la expresión de que el antecedente es una condición suficiente (pero no necesaria) para el consecuente: una sentencia condicional sólo deja de satisfacerse en el caso de que el antecedente sea verdadero y el consecuente falso. Por ejemplo, en nuestra conceptualización de los padres y los antepasados, y con la interpretación definida por *i*, la sentencia

$$(\forall x) (\forall y) (P(x, y) \vee M(x, y) \rightarrow T(x, y))$$

se satisface para $A(x)=\text{José}, A(y)=\text{Ana}$, puesto que $(\text{José}, \text{Ana}) \in \text{Padre}$ y $(\text{José}, \text{Ana}) \in \text{Antepasado}$, y por tanto se satisfacen el antecedente y el consecuente. Pero también se satisface para $A(x)=\text{Mortadelo}, A(y)=\text{Filemón}$, puesto que

(Mortadelo, Filemón) no está ni en la relación "Padre" ni en la relación "Madre" ni en la relación "Antepasado", y asimismo se satisface para $A(x)=\text{José}$, $A(y)=\text{Luis}$, ya que esta asignación no satisface a $P(x,y)$ ni a $M(x,y)$, aunque sí a $T(x,y)$. De hecho, con la interpretación dada, la sentencia se satisface siempre, puesto que todas las parejas de la relación "Padre" y todas la de "Madre" están también en la relación "Antepasado". Como veremos enseguida, en este caso se dice que esa interpretación es un "modelo" de la sentencia.

Aunque ya lo hemos explicado con detalle en el apartado 1.3 del capítulo 2, conviene insistir en que no debe confundirse una sentencia condicional con una "implicación lógica": esta última corresponde, como también veremos (en el apartado 4.2), a un condicional que se satisface *siempre* (para toda i y toda A).

3.3. Inconsistencia y validez

Definición 3.3.1. Una sentencia S es *satisfactible* si y sólo si existen una i y una A tales que $\models_{iA} S$; en caso contrario es *insatisfactible*. Un conjunto de sentencias $\{S_1, S_2, \dots, S_n\}$ es satisfactible si y sólo si existe una i y una A tales que $\models_{iA} (S_1 \wedge S_2 \wedge \dots \wedge S_n)$; en caso contrario se dice que las sentencias son *inconsistentes*. El ejemplo más sencillo de sentencias inconsistentes es $\{P(x), \neg P(x)\}$.

Definición 3.3.2. Una sentencia S es *válida* si y sólo si, para toda i y toda A , $\models_{iA} S$. En este caso se escribe " $\models S$ ". El ejemplo más sencillo de sentencia válida es $P(x) \vee \neg P(x)$ ("principio del tercio excluso"). Una sentencia válida que no contiene variables se llama *tautología*; por ejemplo, $P(a) \vee \neg P(a)$.

Los conceptos de completitud y consistencia de un sistema axiomático son los mismos definidos en el capítulo 2 (apartado 3.4). La extensión del concepto de equivalencia entre sentencias (apartado 3.5) la haremos más adelante, junto con la definición de implicación lógica. Veamos ahora un concepto propio de la lógica de predicados, el de "modelo", que es importante en las aplicaciones de la lógica a las bases de datos deductivas.

3.4. Modelos

Las sentencias que escribimos para representar hechos o relaciones de una conceptualización no son sentencias válidas:² se satisfacen para una determinada interpretación, pero no para otras.

² Una sentencia válida, al satisfacerse siempre, no representa nada. Desde el punto de vista de la teoría de la información, no proporciona información alguna.

Definición 3.4.1. Una interpretación i es un *modelo de una sentencia* S si S se satisface con esa i para todas las asignaciones posibles de sus variables; se escribe " $\models_i S$ ". Y una interpretación es un *modelo de un conjunto de sentencias* si es un modelo de todas y cada una de ellas.

Ya hemos visto más arriba que la interpretación "natural" en la conceptualización de los "Padres", "Madres" y "Antepasados" es un modelo de la sentencia

$$(\forall x) (\forall y) (P(x, y) \vee M(x, y) \rightarrow T(x, y))$$

y también podríamos ver que es un modelo de la sentencia que completa la definición de "antepasado":

$$(\forall x) (\forall y) (\forall z) ((P(x, z) \vee M(x, z)) \wedge T(z, y)) \rightarrow T(x, y)$$

Veamos otro ejemplo. Consideremos una conceptualización con tres individuos, IndA, IndB, IndC y tres propiedades aplicables a estos individuos: ser político, ser mentiroso e ir al infierno. Formalizaremos los individuos con los símbolos " a ", " b ", " c ", es decir: $i(a)=\text{IndA}$, $i(b)=\text{IndB}$, $i(c)=\text{IndC}$ (en todas las interpretaciones que vamos a considerar). Representaremos las propiedades con los símbolos de predicado (de grado 1) " P ", " M " y " F ". Sean las sentencias:

$$S1: (\forall x) (P(x) \rightarrow M(x))$$

$$S2: (\forall x) (M(x) \rightarrow F(x))$$

("todos los políticos son mentirosos" y "todos los mentirosos van al infierno").

La interpretación i_1 tal que:

$$i_1(P) = \{\text{IndA}\}$$

$$i_1(M) = \{\text{IndA}\}$$

$$i_1(F) = \{\text{IndA}\}$$

("sólo el individuo IndA es político, mentiroso y va al infierno; los otros dos no tienen ninguna de esas propiedades") es un modelo de las sentencias. En efecto, para la asignación $A(x)=a$ las dos sentencias se satisfacen, puesto que se satisfacen sus antecedentes ($P(a)$ y $M(a)$) y sus consecuentes ($M(a)$ y $F(a)$); para la asignación $A(x)=b$ y para la asignación $A(x)=c$ también se satisfacen, puesto ni los antecedentes ni los consecuentes se satisfacen.

La interpretación i_2 tal que:

$$i_2(P) = \{\text{IndA}\}$$

$$i_2(M) = \{\text{IndA}, \text{IndB}\}$$

$$i_2(F) = \{\text{IndA}, \text{IndB}, \text{IndC}\}$$

("IndA es político, mentiroso y va al infierno; IndB no es político, pero es mentiroso y va al infierno; IndC no es político ni mentiroso, pero va al infierno") también es un modelo: para $A(x)=a$ se satisfacen antecedentes y consecuentes de $S1$ y $S2$ (luego ambas se satisfacen); para $A(x)=b$ no se satisface el antecedente de $S1$ (luego $S1$ se satisface) y se satisfacen el antecedente y el consecuente de $S2$ (luego $S2$ se satisface); para $A(x)=c$ no se satisfacen los antecedentes (luego $S1$ y $S2$ se satisfacen).

Sin embargo, la interpretación i_3 tal que:

$$i_3(P) = \{\text{IndA}\}$$

$$i_3(M) = \{\text{IndA}\}$$

$$i_3(F) = \emptyset$$

("sólo el individuo A es político, sólo él es mentiroso, y nadie va al infierno") no es un modelo: para $A(x)=a$, $S1$ se satisface, pero no $S2$, porque su antecedente se satisface y no su consecuente.

Tampoco es un modelo la interpretación i_4 tal que:

$$i_4(P) = \{\text{IndA}\}$$

$$i_4(M) = \emptyset$$

$$i_4(F) = \emptyset$$

ya que en este caso se satisface $S2$ pero no $S1$.

Se puede decir que las sentencias condicionales cerradas definen *mundos posibles*: i_1 e i_2 son dos de los modelos de $S1$ y $S2$, o mundos definidos por $S1$ y $S2$.

Supongamos que a $S1$ y $S2$ añadimos

$$S3: P(a)$$

i_1 e i_2 siguen siendo modelos de $\{S1, S2, S3\}$, pero i_1 es el *modelo mínimo*. Un modelo es mínimo si ningún subconjunto de él es un modelo. Si no hubiésemos añadido $S3$, el modelo mínimo sería i_0 tal que $i_0(P) = i_0(M) = i_0(F) = \emptyset$.

4. Sistemas inferenciales

4.1. Razonamientos en lógica de predicados

Ahora extenderemos a la lógica de predicados todo lo que hemos estudiado en el apartado 5 del capítulo 2 para la de proposiciones. Como antes, sugerimos al lector que vaya repasando aquél a medida que avanza en éste.

De momento, todo lo dicho en el apartado 5.1 sigue siendo aplicable, con dos salvedades:

- Lo que allí eran "variables proposicionales" ahora serán "fórmulas atómicas".
- El método de las "tablas de verdad" es ahora inaplicable.

4.2. Implicación lógica y razonamientos deductivos

Definición 4.2.1. Un conjunto de sentencias $\{S_1, S_2, \dots, S_n\}$ *implica lógicamente* a una sentencia S si y sólo si toda iA (*interpretación y asignación de variables*) que satisface al conjunto satisface también a S . Esto se expresa con la notación:

$$\{S_1, S_2, \dots, S_n\} \models S$$

o lo que es igual (por la definición de la semántica de la conjunción y del condicional):

$$\models (S_1 \wedge S_2 \wedge \dots \wedge S_n \rightarrow S)$$

El teorema 5.2.2 del capítulo 2 tiene también su correspondiente, pero pasemos ya a lo que más nos interesa: los razonamientos.

Definición 4.2.2. Un razonamiento formado por las premisas P_1, P_2, \dots, P_n y la conclusión C es un *razonamiento válido o deductivo* si y sólo si las premisas implican lógicamente a la conclusión:

$$\models (P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow C)$$

También podemos extender la definición de equivalencia (que en el capítulo 2 hacíamos en el apartado 3.5):

Definición 4.2.3. S_1 y S_2 son *sentencias equivalentes* si y sólo si $S_1 \models S_2$ y $S_2 \models S_1$, o, lo que es lo mismo, $\models (S_1 \leftrightarrow S_2)$. Expresaremos este hecho con la notación: $S_1 \equiv S_2$.

De la definición de *sentencia válida* y de *modelo* se sigue que $\models S$ si y sólo si para toda i ocurre que $\models_i S$ (S es una sentencia válida si y sólo si toda interpretación es un modelo de S). Por tanto, si $S1$ implica a $S2$ deberá ocurrir que, para toda i , $\models_i (S1 \rightarrow S2)$. Como el condicional sólo deja de satisfacerse en el caso de que se satisfaga el antecedente y no el consecuente, resulta que decir que $S1$ implica a $S2$ es lo mismo que decir que toda i que sea modelo de $S1$ también lo es de $S2$, o que el conjunto de modelos de $S2$ es un subconjunto del de $S1$. Prosiguiendo el razonamiento³, es fácil concluir que $S1$ y $S2$ son equivalentes si y sólo si tienen el mismo conjunto de modelos: todo modelo de $S1$ lo es de $S2$ y viceversa.

Las equivalencias dadas en el capítulo 2 (las que se derivan de las leyes con forma de bicondicional que aparecen en los apartados 2.3, 2.4 y 2.5) siguen siendo válidas. Repetimos algunas de ellas (junto con dos nuevas: las que permiten intercambiar los cuantificadores existencial y universal):

$$(1) \neg(\neg S) \equiv S$$

$$(2) \neg(S1 \vee S2) \equiv \neg S1 \wedge \neg S2$$

$$(3) \neg(S1 \wedge S2) \equiv \neg S1 \vee \neg S2$$

$$(4) S1 \vee (S2 \wedge S3) \equiv (S1 \vee S2) \wedge (S1 \vee S3)$$

$$(5) S1 \wedge (S2 \vee S3) \equiv (S1 \wedge S2) \vee (S1 \wedge S3)$$

$$(6) (S1 \rightarrow S2) \equiv (\neg S1 \vee S2)$$

$$(7) (S1 \leftrightarrow S2) \equiv (S1 \rightarrow S2) \wedge (S2 \rightarrow S1) \equiv (\neg S1 \vee S2) \wedge (\neg S2 \vee S1)$$

$$(8) (\forall x) (S) \equiv \neg(\exists x) (\neg S)$$

$$(9) (\exists x) (S) \equiv \neg(\forall x) (\neg S)$$

$$(10) (\neg S1 \rightarrow (S2 \wedge \neg S2)) \equiv S1$$

La última proviene de la *ley de reducción al absurdo* (apartado 2.5.6 del capítulo 2): si de la negación de $S1$ se sigue una contradicción entonces puede afirmarse $S1$ (y viceversa: si $S1$ se satisface entonces de su negación resultará una contradicción).

³ Obsérvese que si quisiéramos formalizar estos razonamientos habríamos de hacerlo en lógica de predicados de orden superior (capítulo 5, apartado 2.1): " \models ", " \equiv ", etc., son predicados sobre sentencias que contienen predicados.

Hagamos un ejercicio de transformaciones sucesivas de una sentencia mediante aplicación de estas equivalencias. Tomemos la regla recursiva de la definición de "Antepasado" (suponiendo, como hacíamos en el ejemplo 1.7.5, que "P" representa "padre o madre", para que la sentencia sea más manejable):

$$\begin{aligned}
 & (\forall x) (\forall y) (\forall z) (P(x, z) \wedge T(z, y) \rightarrow T(x, y)) \\
 \equiv & (\forall x) (\forall y) (\forall z) (\neg(P(x, z) \wedge T(z, y)) \vee T(x, y)) \quad (\text{por (6)}) \\
 \equiv & (\forall x) (\forall y) (\forall z) ((\neg P(x, z) \vee \neg T(z, y)) \vee T(x, y)) \quad (\text{por (3)}) \\
 \equiv & (\forall x) (\forall y) ((\forall z) (\neg P(x, z) \vee \neg T(z, y)) \vee T(x, y)) \quad (\text{porque } z \text{ no} \\
 & \text{afecta a } T(x, y)) \\
 \equiv & (\forall x) (\forall y) (\neg(\exists z) (\neg(\neg P(x, z) \vee \neg T(z, y))) \vee T(x, y)) \quad (\text{por (8)}) \\
 \equiv & (\forall x) (\forall y) (\neg(\exists z) (P(x, z) \wedge T(z, y)) \vee T(x, y)) \quad (\text{por (2) y por (1)}) \\
 \equiv & (\forall x) (\forall y) ((\exists z) (P(x, z) \wedge T(z, y)) \rightarrow T(x, y)) \quad (\text{por (6)})
 \end{aligned}$$

Tras estas prosaicas transformaciones hemos llegado a un resultado interesante que, generalizándolo, podemos enunciar así: *si una variable figura en el antecedente de un condicional y no en su consecuente, entonces es igual cuantificar la variable universalmente sobre el conjunto de la sentencia que cuantificarla existencialmente sobre el antecedente*. Normalmente, la segunda forma se corresponde mejor con la expresión en lenguaje natural: en este ejemplo, "si existe algún individuo z tal que x es padre de z y z es antepasado de y , entonces x es antepasado de y ".

La sentencia escrita en la tercera línea está en *forma clausulada*. Esta forma la habíamos definido en el apartado 5.5 del capítulo 2 para la lógica de proposiciones, y en el apartado 4.5 de éste extenderemos la definición a la lógica de predicados.

4.3. Reglas de inferencia

Puesto que las leyes de la lógica de proposiciones siguen siendo válidas en lógica de predicados, las reglas de inferencia también lo son (como siempre, sustituyendo "variable proposicional" por "fórmula atómica"). En particular, el lector debe recordar las reglas de *modus ponens*, *modus tollens*, *introducción de conjunción* y *eliminación de conjunción* (capítulo 2, apartado 5.3).

Una regla propia ya de la lógica de predicados es la regla de *particularización de un universal*, fundamentada en la ley de especificación (axioma A4 del apartado 2.5.1), que permite razonar de lo general a lo particular:

$$\frac{P: (\forall x) (S(x))}{C: S(a)}$$

Como ejemplo de aplicación de estas reglas, consideremos el razonamiento del ejemplo 1.7.1:

P1: Todos los libros de informática son terriblemente aburridos

P2: Este es un libro de informática

C: Este libro es terriblemente aburrido

que habíamos formalizado así:

$P1: (\forall x) (I(x) \rightarrow A(x))$

$P2: I(e)$

$C: A(e)$

Pues bien, este razonamiento se obtiene aplicando primero la regla de particularización de un universal a *P1* (particularizando para $x=e$) y luego la regla de *modus ponens* al resultado y a *P2*.

Este otro:

P1: Todos los libros de informática son terriblemente aburridos

P2: Este libro no es terriblemente aburrido

C: Este no es un libro de informática

es decir:

$P1: (\forall x) (I(x) \rightarrow A(x))$

$P2: \neg A(e)$

$C: \neg I(e)$

se obtiene mediante aplicación sucesiva de las reglas de particularización de un universal y de *modus tollens*.

Hay más reglas de inferencia, pero comentaremos solamente, para terminar, la *regla de particularización de un existencial*:

$P: (\exists x) (S(x))$

$C: S(a)$

que utilizaremos más adelante, y que requiere un comentario especial. En efecto, hay que advertir que ahora no es cierto que $\vdash ((\exists x) (S(x)) \rightarrow S(a))$ ⁴. El razonamiento únicamente puede justificarse considerando que la constante "*a*" que sustituye a la variable cuantificada "*x*" no es como las demás constantes de la conceptualización: no es un valor concreto de entre los que puede tomar *x*, sino un valor arbitrario. Se dice que es una *constante de Skolem*. Por ejemplo, "existe al menos un individuo que es autor de este libro" se puede formalizar como:

$(\exists x) (A(x, e))$

⁴ Recuérdese que, según el teorema 5.4.3 del capítulo 2, para que un sistema inferencial sea consistente, todas sus reglas de inferencia tienen que corresponderse con tesis del sistema axiomático.

La regla dice que se puede obtener la conclusión

$$A(i, e)$$

donde " i " es una constante arbitraria que no puede coincidir con ninguna de las utilizadas en la conceptualización. Informalmente el razonamiento es así: "la sentencia dice que existe un individuo (al menos) que cumple ciertas condiciones; inventemos un nombre para este individuo y sigamos adelante".

Ahora bien, si la sentencia abarcada por el cuantificador existencial que se elimina está dentro del alcance de un cuantificador universal entonces no podemos sustituir la variable por una constante de Skolem. Para ver por qué es así, consideremos este otro ejemplo: "todos los libros tienen al menos un autor", o "para todo x , si x es un libro, entonces existe un y tal que y es el autor de x ", o:

$$(\forall x) (L(x) \rightarrow (\exists y) A(y, x))$$

Si sustituimos y por una constante de Skolem, i , obtenemos:

$$(\forall x) (L(x) \rightarrow A(i, x))$$

que dice "para todo x , si x es un libro entonces el individuo representado por i es su autor (o uno de sus autores)", es decir, "el individuo representado por i es autor de todos los libros", que, obviamente, no es lo que pretende expresar la primera sentencia. Lo que ocurre es que el valor de la variable y (el autor) depende del que tome x (el libro), y, por tanto, en lugar de sustituirla por una constante debe sustituirse por una *función de Skolem*:

$$(\forall x) (L(x) \rightarrow A(\overline{i(x)}, x))$$

("todo libro, x , tiene al menos un autor, el representado por " $\overline{i(x)}$ ").

En general, la regla de particularización de un existencial es:

$$\frac{P: (\exists x) (S(x, y_1, y_2, \dots, y_n))}{C: S(f(y_1, y_2, \dots, y_n), y_1, y_2, \dots, y_n)}$$

donde f es un símbolo de función diferente de los de todas las funciones que aparezcan en la conceptualización.

4.4. Sistemas inferenciales

Las definiciones y el teorema enunciados en el apartado 5.4 del capítulo 2 son muy generales. Todo lo que allí se dice es aplicable en su integridad a la lógica de predicados.

Lo que veíamos en los apartados siguientes de aquél capítulo (forma clausulada, resolución y refutación) requiere algunas extensiones para aplicarse a la lógica de predicados.

4.5. Forma clausulada de la lógica de predicados

Definición 4.5.1. Una *cláusula* es una sentencia de la forma:

$$L_1 \vee L_2 \vee \dots \vee L_n$$

donde los L_i son literales (fórmulas atómicas con o sin el símbolo " \neg ") con cualquier número de variables cada uno. Todas las variables se suponen cuantificadas universalmente, aunque no se escriba $(\forall x_1) (\forall x_2) \dots$ delante de la cláusula.

Definición 4.5.2. Diremos que una sentencia está en *forma clausulada* si tiene la forma:

$$(L_{11} \vee L_{12} \vee \dots) \wedge (L_{21} \vee L_{22} \dots) \wedge \dots$$

en la que cada cláusula $(L_{i1} \vee L_{i2} \vee \dots)$ tiene sus propias variables.

De una cláusula se dice también que es una *colección de literales* (implícitamente unidos por disyunciones), y de una sentencia en forma clausulada, que es una *colección de cláusulas* (implícitamente unidas por conjunciones).

Se observará que, a partir de este momento, suponemos que no intervienen variables proposicionales en las sentencias de la lógica de predicados (pero, si existieran, bastaría tratarlas igual que a fórmulas atómicas en las que sólo interviniesen constantes).

Teorema 4.5.3. Para toda sentencia de la lógica de predicados existe una sentencia equivalente en forma clausulada.

La demostración es constructiva: vamos a ver un procedimiento, ampliación del estudiado para la lógica de proposiciones, que permite pasar a forma clausulada cualquier sentencia. Los pasos, ahora, son siete:

1. Eliminación de todas las conectivas que no sean " \vee " o " \wedge " (normalmente, condicionales y bicondicionales). Se hace igual que en lógica de proposiciones:

$$(A \rightarrow B) \equiv (\neg A \vee B)$$

$$(A \leftrightarrow B) \equiv (\neg A \vee B) \wedge (A \vee \neg B)$$

2. Introducción de negaciones. Se seguirán utilizando las leyes de de Morgan y la ley de doble negación:

$$\neg(A \vee B) \equiv (\neg A \wedge \neg B)$$

$$\neg(A \wedge B) \equiv (\neg A \vee \neg B)$$

$$\neg\neg A \equiv A$$

y ahora, además, las relaciones ya conocidas entre los cuantificadores:

$$\neg(\exists x) (A) \equiv (\forall x) (\neg A)$$

$$\neg(\forall x) (A) \equiv (\exists x) (\neg A)$$

3. Independización de las variables cuantificadas. Se cambian los nombres de las variables que sean necesarios para que cada cuantificador se refiera a su propia variable. Por ejemplo,

$$(\forall x) (\neg P(x) \vee (\exists x) (Q(x))) \equiv (\forall x) (\neg P(x) \vee (\exists y) (Q(y)))$$

4. Eliminación de los cuantificadores existenciales. Este es el paso de más difícil justificación teórica. En efecto, está basado en la "equivalencia"

$$(\exists x) (P(x)) \equiv P(a)$$

que es de todo punto incorrecta. En efecto, podemos asegurar que

$$\vdash (P(a) \rightarrow (\exists x) (P(x)))$$

porque es la "ley de inespecificación" que hemos demostrado como "teorema 1" en el apartado 2.5.4. Pero la sentencia

$$(\exists x) (P(x)) \rightarrow P(a)$$

no es válida. Ahora bien, recordemos la discusión del apartado 4.3 sobre la "regla de particularización de un existencial": consideramos que " a " no es una constante ordinaria que representa a un individuo concreto del universo del discurso, sino una *constante de Skolem*, que representa a "algún individuo" (indeterminado) y que nos permite prescindir de " \exists ". Recordemos también que si x está dentro del alcance de un cuantificador universal entonces hay que introducir una *función de Skolem*.

5. Eliminación de los cuantificadores universales. Contra lo que pudiera pensarse, este paso no plantea ningún problema. En este momento, si se han seguido los pasos anteriores, sólo existen cuantificadores universales, y cada uno de ellos afecta a una variable diferente. Por tanto, podemos escribirlos todos al comienzo de la sentencia. Y, puesto que todas las variables están universalmente cuantificadas (como ya hemos dicho, sólo consideramos sentencias cerradas), podemos omitir la escritura de tales cuantificadores. No se trata, pues, en rigor, de "eliminarlos" sino de suponer que siempre existen.

Para centrar las ideas sobre los pasos seguidos hasta ahora, supongamos que partimos de la sentencia

$$[(\forall x) (P(x))] \rightarrow [(\forall x) (\forall y) (\exists z) (P(x,y,z) \rightarrow (\forall u) (R(x,y,z,u)))]$$

Eliminando los condicionales,

$$[\neg(\forall x) (P(x))] \vee [(\forall x) (\forall y) (\exists z) (\neg P(x,y,z) \vee (\forall u) (R(x,y,z,u)))]$$

Introduciendo negaciones (la única a introducir es la primera),

$$[(\exists x) (\neg P(x))] \vee [(\forall x) (\forall y) (\exists z) (\neg P(x,y,z) \vee (\forall u) (R(x,y,z,u)))]$$

Independizando las dos variables que tiene el nombre "x",

$$[(\exists x) (\neg P(x))] \vee [(\forall w) (\forall y) (\exists z) (\neg P(w,y,z) \vee (\forall u) (R(w,y,z,u)))]$$

La variable x puede sustituirse por una constante de Skolem. Pero la z está dentro del alcance de dos cuantificadores universales: el de w y el de y . Por tanto, la sustituiremos por una función de Skolem, $f(w,y)$:

$$[\neg P(a)] \vee [(\forall w) (\forall y) (\neg P(w,y,f(w,y)) \vee (\forall u) (R(w,y,f(w,y),u)))]$$

Por último, ponemos en cabeza todos los cuantificadores universales:

$$(\forall w) (\forall y) (\forall u) \{ [\neg P(a)] \vee [\neg P(w,y,f(w,y)) \vee R(w,y,f(w,y),u)] \}$$

Y considerando la asociatividad de la disyunción, y dando por implícitamente cuantificadas universalmente a todas las variables, escribiremos:

$$\neg P(a) \vee \neg(w,y,f(w,y)) \vee R(w,y,f(w,y),u)$$

6. Distribución de " \wedge " sobre " \vee ". En el último ejemplo, ya hemos llegado a una disyunción de literales, o sea, a una cláusula. Pero, en general, tras el último paso tendremos una sentencia formada por literales unidos por las conectivas " \wedge " y " \vee ", y aplicaremos, igual que hacíamos en lógica de proposiciones, la propiedad distributiva

$$(L_1 \wedge L_2) \vee L_3 \equiv (L_1 \vee L_3) \wedge (L_2 \vee L_3)$$

para llegar a una conjunción de cláusulas.

7. Redenominación de variables para que cada cláusula tenga las suyas propias. Este paso tiene su justificación en la ley de distribución del cuantificador universal sobre la conjunción. En efecto, si tras los pasos anteriores se ha llegado a una sentencia como

$$P(x) \wedge Q(x)$$

(dos cláusulas constituidas cada una por un literal), en realidad la sentencia es:

$$(\forall x) (P(x) \wedge Q(x))$$

y lo que hacemos es sustituirla por su equivalente

$$(\forall x) (P(x)) \wedge (\forall y) (P(y))$$

es decir, nos quedamos con las cláusulas $P(x)$ y $P(y)$.

Ejemplo:

Sigamos todos los pasos con la siguiente sentencia:

$$(\forall x) \{ [(\exists y) (P(y) \rightarrow Q(x,y) \wedge R(y))] \wedge [\neg(\exists y) (P(f(y)) \rightarrow Q(x,y))] \}$$

$$1: (\forall x) \{ [(\exists y) (\neg P(y) \vee (Q(x,y) \wedge R(y)))] \wedge [\neg(\exists y) (\neg P(f(y)) \vee Q(x,y))] \}$$

$$2: (\forall x) \{ [(\exists y) (\neg P(y) \vee (Q(x,y) \wedge R(y)))] \wedge [(\forall y) (P(f(y)) \wedge \neg Q(x,y))] \}$$

$$3: (\forall x) \{ [(\exists y) (\neg P(y) \vee (Q(x,y) \wedge R(y)))] \wedge [(\forall z) (P(f(z)) \wedge \neg Q(x,z))] \}$$

$$4: (\forall x) \{ [\neg P(g(x)) \vee (Q(x,g(x)) \wedge R(g(x)))] \wedge [(\forall z) (P(f(z)) \wedge \neg Q(x,z))] \}$$

$$5: [\neg P(g(x)) \vee (Q(x,g(x)) \wedge R(g(x)))] \wedge [P(f(z)) \wedge \neg Q(x,z)]$$

$$6: [\neg P(g(x)) \vee Q(x,g(x))] \wedge [\neg P(g(x)) \vee R(g(x))] \wedge [P(f(z))] \wedge [\neg Q(x,z)]$$

- 7: Resultan, finalmente, cuatro cláusulas:

$$\neg P(g(x_1)) \vee Q(x_1, g(x_1))$$

$$\neg P(g(x_2)) \vee R(g(x_2))$$

$$P(f(z_1))$$

$$\neg Q(x_3, z_2)$$

Las cláusulas se pueden expresar también como sentencias condicionales, igual que veíamos en el apartado 5.6 del capítulo 2: el antecedente es la conjunción de los literales negativos, hechos positivos, y el consecuente es la disyunción de los literales positivos. Y, lo mismo que allí, se llaman cláusulas de Horn a las que sólo tienen un literal positivo ("cabeza") o ninguno, y cláusula vacía, \emptyset , a aquella en la que han desaparecido todos los literales.

Con las sentencias en forma clausulada, la idea de la resolución es la misma que en lógica de proposiciones: dadas dos generatrices que comparten un literal positivo en la una y negativo en la otra, obtener la resolvente eliminando ese literal y conservando los demás. Pero ahora hay un detalle nuevo a considerar: como los literales dependen de argumentos, habrá que igualar los argumentos de los literales que se eliminan. Por ejemplo, consideremos la inferencia de *modus ponens*:

$$\frac{(\forall x) (H(x) \rightarrow M(x)) \quad H(s)}{M(s)}$$

Poniendo las sentencias en forma clausulada,

$$\frac{\neg H(x) \vee M(x) \quad H(s)}{M(s)}$$

Si eliminamos $H(s)$ es porque podemos sustituir x por s en la primera premisa, y así "*unificar*" los dos literales. El proceso de unificación está fundamentado en la ley de especificación (como la regla de inferencia de particularización de un universal explicada en el apartado 4.3), pero no es siempre tan fácil como en este ejemplo. Vamos a estudiarlo de manera más general.

4.6. Sustitución y unificación

Ya hemos definido la sustitución en el apartado 2.5.2. Pero ahora, por una parte, no tenemos variables proposicionales, y, por otra, sólo nos interesa sustituir variables por términos, y sólo en los literales, no en las sentencias en general. Nos quedaremos, pues, con una definición restringida de sustitución:

Definición 4.6.1. Dadas unas variables x_1, x_2, \dots, x_n y unos términos t_1, t_2, \dots, t_n en los que no figuran esas variables, llamaremos ahora *sustitución* a un conjunto de pares ordenados

$$s = \{t_1/x_1, t_2/x_2, \dots, t_n/x_n\}$$

Definición 4.6.2. Dados un literal L que contiene las variables x_1, x_2, \dots, x_n , y una sustitución s (cuyos términos no pueden contener símbolos constantes ni de función que ya estén en L), la *operación de sustitución* consiste en poner t_1 en todos los lugares de L donde aparezca x_i , y ello para todos los pares t_i/x_i de s . El resultado, que se representa por Ls , es un *caso de sustitución en L* .⁵

Por ejemplo, sean: $L = P(a, x, f(y))$

$$s_1 = \{b/x, c/y\}$$

$$s_2 = \{b/x, g(z)/y\}$$

$$s_3 = \{z/x, w/y\}$$

Los tres casos respectivos de sustitución en L serán:

$$Ls_1 = P(a, b, f(c))$$

$$Ls_2 = P(a, b, f(g(z)))$$

$$Ls_3 = P(a, z, f(w))$$

Ls_1 es un *caso terminal*, o un *ejemplar*⁶, de L : llamaremos así a los que no contienen variables. Ls_3 es una *variante alfabética de L* : sólo se han cambiado los nombres de las variables por otros.

Definición 4.6.3. Dadas dos sustituciones, s_1 y s_2 , su *composición*, s_1s_2 , es una sustitución tal que $Ls_1s_2 = (Ls_1)s_2$.

La composición de sustituciones es asociativa ($(s_1s_2)s_3 = s_1(s_2s_3)$) pero, en general, no es conmutativa $s_1s_2 \neq s_2s_1$. La razón es que s_1 y s_2 pueden incluir variables idénticas que se sustituyen por términos diferentes, por lo que el resultado de la composición dependerá de cuál se aplique primero. Por la misma razón, no puede calcularse la composición uniendo simplemente los conjuntos s_1 y s_2 . Para el cálculo de s_1s_2 hemos de aplicar primero s_2 a los términos de s_1 y después añadir los pares de s_2 cuyas variables no están entre las de s_1 . Por ejemplo, la composición de

$$s_1 = \{f(x, y)/z\}$$

y

$$s_2 = \{a/x, b/y, c/z, d/u\}$$

⁵ A veces se utiliza (incorrectamente) la palabra "instancia" (del inglés "instance").

⁶ "ground instance", en inglés.

sería:

$$s_1 s_2 = \{f(a, b) / z, a / x, b / y, d / u\}$$

mientras que, si invertimos el orden,

$$s_2 s_1 = \{a / x, b / y, c / z, d / u\} = s_2$$

(porque cuando fuésemos a aplicar s_1 ya habríamos aplicado antes s_2 , con la que habría desaparecido la z , sustituida por c).

Definición 4.6.4. Diremos que un conjunto de literales $\{L_i\}$ ($i=1,2,\dots,n$) es *unificable* si existe una sustitución s tal que $L_1 s = L_2 s = \dots = L_n s$. Para este caso, diremos que s es un *unificador* de $\{L_i\}$ y que los literales L_i se unifican en $L_i s$.

Por ejemplo, sea $\{L_i\} = \{P(a, x, f(y)), P(a, x, f(b))\}$. Un unificador sería $s = \{c/x, y/b\}$, con el que ambos literales se unifican en $P(a, c, f(b))$. Pero este no es el único: hay otro unificador aún más pequeño, $\mu = \{y/b\}$, con el que se unifican en $P(a, x, f(b))$.

Teorema 4.6.5. Si $\{L_i\}$ es unificable, entonces existe un *unificador más general*, o *unificador mínimo*, μ , que tiene dos propiedades:

a) Si s es otro unificador de $\{L_i\}$ entonces existe una sustitución s' tal que $s = \mu s'$, es decir, $L_i s$ es un caso de $L_i \mu$. (En el ejemplo anterior, $P(a, c, f(b))$ es un caso de $P(a, x, f(b))$ para $s' = \{c/x\}$).

b) $L_i \mu$ es único salvo por variantes alfabéticas.

No entraremos en la demostración de este teorema, que es algo laboriosa, y que es constructiva: se llega a un algoritmo para encontrar el unificador más general de cualquier conjunto de literales que sea unificable.

4.7. La regla de resolución

Sean dos generatrices,

$$G_1 = L_{11} \vee L_{12} \vee \dots$$

$$G_2 = L_{21} \vee L_{22} \vee \dots$$

o, expresadas como conjuntos de literales,

$$G_1 = \{L_{1i}\}$$

$$G_2 = \{L_{2i}\}$$

Y sean $\{l_{1i}\} \subset \{L_{1i}\}$ y $\{l_{2i}\} \subset \{L_{2i}\}$ tales que $\{l_{1i}\} \cup \{\neg l_{2i}\}$ es unificable, siendo μ el unificador mínimo. Entonces decimos que $\{L_{1i}\}$ y $\{L_{2i}\}$ se resuelven en l_{1i} y que de ambas generatrices se infiere la resolvente:

$$[\{L_{1i}\} - \{l_{1i}\}] \mu \cup [\{L_{2i}\} - \{l_{2i}\}] \mu$$

La justificación teórica de esta regla de inferencia es una generalización de las leyes de inferencia de la alternativa que veíamos en el apartado 2.5.5. Para comparar aquella ley con lo que acabamos de decir, obsérvese que, al unificar, l_{1i} se reduce a un solo literal, que corresponde al que allí llamábamos $P(x_1, \dots, x_n)$, y $\{l_{2i}\}$ a otro, $\neg P(x_1, \dots, x_n)$.

Ejemplo:

$$\text{Sean } G_1 = P(a, x, f(a)) \vee \neg Q(x)$$

$$G_2 = \neg P(a, y, f(a)) \vee \neg P(a, y, f(z)) \vee Q(z)$$

Podemos elegir $\{l_{1i}\}$ y $\{l_{2i}\}$ de varios modos, y con cada uno obtendremos una resolvente distinta:

$$\text{a) } \{l_{1i}\} = \{P(a, x, f(a))\}; \{l_{2i}\} = \{\neg P(a, y, f(a))\}$$

El unificador mínimo es $\mu = \{x/y\}$, y la resolvente:

$$R = \neg Q(x) \vee \neg P(a, x, f(z)) \vee Q(z)$$

$$\text{b) } \{l_{1i}\} = \{P(a, x, f(a))\}; \{l_{2i}\} = \{\neg P(a, y, f(z))\}$$

$$\mu = \{x/y, a/z\}$$

$$R = \neg Q(x) \vee \neg P(a, x, f(a)) \vee Q(a)$$

$$\text{c) } \{l_{1i}\} = \{P(a, x, f(a))\}; \{l_{2i}\} = \{\neg P(a, y, f(a)), \neg P(a, y, f(z))\}$$

$$\mu = \{x/y, a/z\}$$

$$R = \neg Q(x) \vee Q(a)$$

Las tres anteriores son resoluciones en P . También podemos hacer una resolución en Q :

$$\text{d) } \{l_{1i}\} = \{\neg Q(x)\}; \{l_{2i}\} = \{Q(z)\}$$

$$\mu = \{x/z\}$$

$$R = P(a, x, f(a)) \vee \neg P(a, y, f(a)) \vee \neg P(a, y, f(x))$$

4.8. Refutación

Aquí es totalmente aplicable todo lo dicho en el apartado 5.8 del capítulo 2, porque la ley de reducción al absurdo se aplica también en lógica de predicados. Es decir, si

$$P = P_1 \wedge P_2 \wedge \dots \wedge P_n$$

es una sentencia con todas sus variables cuantificadas universalmente, y lo mismo C , $P \rightarrow C$ será verdadera y, por tanto, C será una conclusión de P_1, P_2, \dots, P_n , si (y sólo si) de la conjunción de P y $\neg C$ resulta una contradicción.⁷ Con la resolución, la contradicción se manifestará por la obtención de la cláusula vacía.

Se puede demostrar que la refutación con resolución y con búsqueda exhaustiva (definición 5.7.2 del capítulo 2) es un sistema inferencial consistente (si se obtiene la cláusula vacía, el razonamiento formado por P y C es válido) y completo (si el razonamiento es válido, se obtiene la cláusula vacía). El problema es que la búsqueda exhaustiva exige ahora comprobar para todas las unificaciones posibles entre todas las cláusulas, y en problemas cuyo tamaño (que depende del número de premisas, de cláusulas en cada premisa y de literales en cada cláusula) empiece a ser elevado se produce el fenómeno llamado "explosión combinatoria": el número de combinaciones a explorar se hace tan elevado que las máquinas actuales no pueden llegar a la solución en un tiempo aceptable.⁸ Por ello, se utilizan otras estrategias para el sistema inferencial, y no todas ellas son completas.

Ejemplo:

Volvamos al ejemplo 1.7.4, y, para simplificar, supongamos que con el predicado $P(x,y)$ representamos "x es el padre o la madre de y". Tendremos entonces dos hechos y la definición de "abuelo de":

$$P1: P(a, l)$$

$$P2: P(j, a)$$

$$P3: (\forall x) (\forall y) (\forall z) (P(x, y) \wedge P(y, z) \rightarrow A(x, z))$$

⁷ Este resultado se obtiene fácilmente sustituyendo $P \rightarrow C \equiv \neg(P \wedge \neg C)$ en la tesis que corresponde a la equivalencia (10) del apartado 4.2 (de la misma forma que hacíamos en el apartado 5.8 del capítulo 2).

⁸ En términos de algorítmica, el algoritmo es de complejidad no polinómica (apartado 3.1 del capítulo 7 del Tema "Algoritmos").

La conclusión evidente en este caso es $A(j,l)$. Pero normalmente tendremos muchos más hechos⁹, y podríamos plantearnos la pregunta: "dados los hechos y la definición de abuelo, ¿existen abuelos?" Vamos a ver que la refutación no sólo nos permite responder a la pregunta, sino, si la respuesta es afirmativa, averiguar los individuos que están en la relación.

La pregunta que acabamos de hacer se formalizaría así:

$$C: (\exists x) (\exists y) (A(x, y))$$

Si la respuesta es positiva entonces se podrá refutar $\neg C$. Pongamos primero $P3$ y $\neg C$ en forma clausulada:

$$P3: \neg P(x, y) \vee \neg P(y, z) \vee A(x, z)$$

$$\neg C: \neg A(u, v)$$

Resolviendo en A estas dos cláusulas mediante $\mu = \{x/u, z/v\}$,

$$C1: \neg P(x, y) \vee \neg P(y, z)$$

Resolviendo en P $P1$ con $C1$ mediante $\mu = \{a/y, l/z\}$,

$$C2: \neg P(x, a)$$

Y resolviendo finalmente $P2$ con $C2$ mediante $\mu = \{j/x\}$,

$$C3: \varnothing$$

Pero no solamente hemos obtenido la refutación, sino también los valores de las variables mediante los cuales se refuta. Efectivamente, las dos variables iniciales de la pregunta, u y v , las sustituimos por x y z , y posteriormente z se sustituyó por l y x por j . Luego l y j son los dos individuos que están en la relación.

Ahora bien, supongamos que nuestro procedimiento es tal que al resolver $P1$ con $C1$ escoge el primer literal de $P1$ en lugar del segundo: En tal caso, haría la unificación con $\mu = \{a/x, l/y\}$ y obtendría:

$$C2: \neg P(l, z)$$

y si el procedimiento (no de búsqueda exhaustiva) ya no vuelve a considerar $C1$, entonces habrá llegado a un punto muerto, puesto que no puede hacer más resoluciones.

⁹ Por ejemplo, dejamos como ejercicio al lector el análisis de lo que ocurre cuando el universo del discurso es el considerado en el ejemplo del apartado 3.1.

5. Resumen

La lógica de predicados nos permite entrar en la composición de los enunciados, que en lógica de proposiciones se representan con simples variables proposicionales. Ahora tendremos, en lugar de variables proposicionales, predicados, que representan propiedades de individuos o relaciones entre individuos, y constantes y variables, que representan a esos individuos.

Hemos seguido un camino paralelo al del capítulo 2, ampliando a la lógica de predicados el sistema axiomático, la semántica y los sistemas inferenciales, deteniéndonos especialmente en la regla de resolución, cuya aplicación es algo más complicada aquí, primero por la existencia de variables cuantificadas, que exige ciertas manipulaciones sobre las sentencias para expresarlas en forma clausulada, y luego porque en el emparejamiento de literales complementarios hay que buscar unificaciones.

El último ejemplo, que se complementa más adelante con el ejercicio 7.4, permite vislumbrar cómo el método de resolución puede servir para búsquedas en bases de datos construidas sobre la declaración de hechos elementales y de definiciones de relaciones.

6. Notas histórica y bibliográfica

Robinson (1965) propuso el método de resolución junto con un algoritmo de unificación. Anteriormente, Davis y Putnam (1960) habían presentado un procedimiento para la expresión de sentencias en forma clausulada. Y mucho antes, el matemático Thoralf Skolem (1928) había propuesto el uso de las funciones que llevan su nombre.

La que hemos llamada "búsqueda exhaustiva" es la estrategia de control más rudimentaria para sistemas inferenciales basados en la regla de resolución. Se han propuesto muchas otras estrategias, algunas no completas en aras de una mayor eficacia. Una buena síntesis puede encontrarse en Nilsson (1982, cap. 5).

Aunque en este capítulo hayamos puesto un énfasis especial en los métodos basados en la regla de resolución, no se debe concluir por ello que éste sea el único sistema inferencial programable. De hecho, hay situaciones en las que la forma clausulada de la lógica no es la más apropiada, y se prefiere un sistema inferencial que opere sobre sentencias expresadas en la forma "estándar".

Aparte de las aplicaciones para el diseño de sistemas basados en conocimiento, de las que hablaremos en el capítulo 6, la lógica formal se utiliza también como base para la definición de lenguajes de especificación de programas. Este es un tema de gran importancia en ingeniería del software: la programación, tradicionalmente, ha tenido siempre más de "arte" que de "ciencia"; ahora se trata

de elaborar teorías y métodos formales que permitan mejorar la productividad y la fiabilidad del software.

Hay muchos libros sobre lógica recomendables para ampliar lo expuesto en este capítulo. Destacamos los de Robinson (1979), Kowalski (1979) y Cuenca (1986), y, muy especialmente, el de Genesereth y Nilsson (1987). A través de la Internet pueden obtenerse materiales didácticos que se utilizan en la Universidad de Stanford para un curso basado en este último texto (Genesereth, 1995).

7. Ejercicios

7.1. Expresar como sentencias en lógica de predicados los siguientes enunciados:

- a) "Existen individuos que, sin ser completamente idiotas, se comportan como tales".
- b) "Todo lo que no es tradición es plagio". (Eugenio D'Ors).
- c) "Diremos que una máquina es inteligente para un tipo de tareas si un observador es incapaz de distinguir por el resultado a la máquina de un humano que sabe resolver ese tipo de tareas". (Esto es lo que se llama "prueba de Turing": véase el apartado 1.1 del capítulo 6).
- d) "Un descendiente de una persona es o bien un hijo de esa persona o bien un hijo de un descendiente".

7.2. Considerar la premisa:

P: En un pueblo hay un barbero que afeita a todas las personas del pueblo que no se afeitan a sí mismas, y sólo a ellas".

¿Se puede inferir la conclusión *C*: "El barbero se afeita a sí mismo"? Si esta conclusión fuese verdadera, la premisa nos dice que tendría que ser falsa (sólo afeita a los que *no* se afeitan a sí mismos), y si fuese falsa, la premisa nos dice que tendría que ser verdadera (afeita a *todos* los que no se afeitan a sí mismos). Esta es la llamada "paradoja del barbero", debida a Bertrand Russell. Descubrir el origen de la paradoja mediante el análisis formal de la premisa.

7.3. Formalizar los siguientes razonamientos y comprobar las conclusiones mediante resolución y refutación:

a) *P1*: Ningún ordenador se equivoca

P2: El que tiene boca se equivoca

C: Ningún ordenador tiene boca

b) *P1*: Algunos ordenadores se equivocan

P2: El que tiene boca se equivoca

C: Algunos ordenadores tienen boca

c) *P1*: Todos los libros de informática son instructivos

P2: Ninguna novela es un libro de informática

C: Ninguna novela es instructiva

d) *P1*: Todos los informáticos saben programar

P2: Los humanistas no saben programar

P3: Algunos humanistas saben leer programas

C: Algunos que saben leer programas no son informáticos

e) *P*: No existe nadie que sea al mismo tiempo maestro de alguien y alumno de ese mismo alguien

C1: Nadie es maestro de sí mismo

C2: Nadie es alumno de sí mismo

f) *P1*: Todos los animales que no cocean son flemáticos

P2: Los asnos no tienen cuernos

P3: Un búfalo siempre puede lanzarlo a uno contra una puerta

P4: Ningún animal que cocea es fácil de engullir

P5: Ningún animal sin cuernos puede lanzarlo a uno contra una puerta

P6: Todos los animales son excitables, excepto los búfalos

C: Los asnos no son fáciles de engullir

g) *P1*: Los animales se irritan siempre mortalmente si no les presto atención

P2: Los únicos animales que me pertenecen están en ese prado

P3: Ningún animal puede adivinar un acertijo a menos que haya sido adecuadamente instruido en un colegio con internado

P4: Ningún animal de los que están en este prado es un tejón

P5: Cuando un animal está mortalmente irritado corre de un lado para otro salvajemente y gruñe

P6: Nunca presto atención a un animal, a no ser que me pertenezca

P7: Ningún animal que haya sido adecuadamente instruido en un colegio con internado corre de un lado para otro salvajemente y gruñe

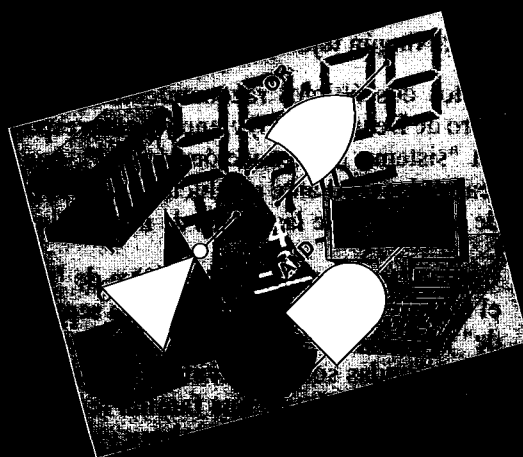
C: Ningún tejón puede adivinar un acertijo

(Estos dos últimos razonamientos proceden de Lewis Carroll. En el libro de Deaño (1986) vienen las correspondientes inferencias mediante un "sistema de deducción natural", que pueden compararse con las realizadas mediante resolución; para el f), Deaño utiliza veintiún pasos de aplicación de las reglas de inferencia, y para el g), veinticuatro).

- 7.4. Completar la definición recursiva de "antepasado de" que dábamos en el ejemplo 1.7.5. para que incluya separadamente las relaciones "padre de" y "madre de". Añadir varios hechos (relaciones "padre de" y "madre de" definidas sobre individuos concretos) imaginarios o reales (tomados, por ejemplo, de una familia real, en cualquiera de las acepciones de "real") de manera que existan varios individuos que estén en la relación "antepasado de". Ver entonces cómo mediante refutación podemos encontrar estas parejas de individuos.

Fundamentos de informática

5



5

Otras lógicas

1. Introducción

La lógica clásica nos permite modelar los razonamientos (en su vertiente funcional, no en la procesal: recuérdese la discusión sobre este asunto en el apartado 1 del capítulo 1). Pero, como todo modelo, no es más que una aproximación a la realidad que se concentra en determinados aspectos relevantes de esa realidad y que no contempla infinidad de matices y de detalles. Las lógicas llamadas "no clásicas" son las que abordan alguno de los aspectos no modelados en la lógica de proposiciones ni en la lógica de predicados. Y son de utilidad en informática cuando tales aspectos juegan un papel importante en el fenómeno o problema a estudiar o en la aplicación a desarrollar.

Dentro de estas "otras lógicas" pueden considerarse dos tipos: uno, el de las que amplían los lenguajes de los cálculos de proposiciones y de predicados y en las que todas las construcciones y teoremas de éstos siguen siendo válidos, y otro, el de las que invalidan algunas leyes. Por ejemplo, en las lógicas multivaloradas o polivalentes, que son aquellas en las que se admiten evaluaciones intermedias entre la "verdad" y la "falsedad", la ley del tercio excluso, $\vdash (A \vee \neg A)$, (o, equivalentemente, $\vdash [\neg(A \wedge \neg A)]$) deja de ser válida.

El campo es demasiado amplio como para pretender darle aquí un tratamiento al mismo nivel formal y con la misma extensión que hemos hecho con las lógicas de proposiciones y de predicados. Nos limitaremos a una presentación resumida

de las lógicas más conocidas, apuntando solamente sus aplicaciones en informática, y nos extenderemos finalmente con un poco más de detalle en una de ellas, la lógica borrosa. Remitimos al lector interesado en estudiar con rigor alguna de estas lógicas a la bibliografía que se comenta en el apartado 6.

2. Ampliaciones de la lógica de predicados

2.1. Lógica de predicados de orden superior

Tras estudiar el capítulo anterior, el lector que haya reparado en su título albergará, seguramente, una duda: ¿por qué "lógica de predicados *de primer orden*"? La respuesta es, naturalmente, que hay lógicas de predicados de segundo orden, de tercero, etc.

En la lógica de predicados de segundo orden los predicados pueden cuantificarse y utilizarse como variables¹. En el lenguaje cotidiano solemos hacerlo. Por ejemplo, cuando decimos "todos los españoles comparten algún rasgo común", que habríamos de formalizar así:

$$(\exists R) ((\forall x) (E(x) \rightarrow R(x)))$$

("existe al menos un rasgo, R , tal que, para todo x , si x es español, x tiene el rasgo R "). O cuando decimos "el cobre es conductor, propiedad que es muy interesante", que formalizaríamos así:

$$C(c) \wedge I(C)$$

Los predicados que se refieren a propiedades (I , en el último ejemplo) pueden también cuantificarse, entrando de este modo en la lógica de predicados de tercer orden, y así sucesivamente.

Las reglas de formación dadas en el capítulo 4 pueden ampliarse fácilmente para que permitan obtener sentencias de la lógica de predicados de orden superior. No ocurre lo mismo con el sistema axiomático. El problema es que muchos sistemas que se han propuesto son inconsistentes y conducen a paradojas. (Por ejemplo, la paradoja de la sentencia "esta sentencia es falsa": si la sentencia es verdadera, hay que concluir que es falsa, y viceversa). Para resolverlo, se han elaborado esquemas teóricos, como las "teorías de los tipos", que en informática encuentran aplicaciones en la definición semántica de lenguajes de programación (tema "Lenguajes", capítulo 5, apartado 4).

¹ En el capítulo anterior lo hemos hecho ocasionalmente, cuando expresábamos propiedades de sentencias, y así lo hicimos observar en las notas a pie de página de los apartados 3.2 y 4.2.

2.2. Lógica de clases y lógica de relaciones

En realidad, las lógicas de clases y de relaciones no añaden nada nuevo a la lógica de predicados: son otra manera de expresarla, con construcciones sintácticas más cercanas a la teoría de conjuntos.

Una *clase* es una entidad abstracta que designa a todos los individuos que comparten alguna propiedad. Conceptualmente, "clase" y "conjunto" son dos cosas diferentes: no es lo mismo la clase de los libros de informática, algo abstracto que designa a todos los objetos que comparten las propiedades de ser libros y de tratar sobre informática, que el conjunto de todos los libros de informática, que es algo concreto. Pero las mismas operaciones que se definen entre conjuntos (unión, intersección, etc.) son también válidas entre clases. Y consiguientemente, el álgebra de Boole, que es un modelo para la teoría de conjuntos y para la lógica de proposiciones, también lo es para la lógica de clases.

El alfabeto de la lógica de clases incluye los símbolos para representar a las clases (A, B, C, \dots), los símbolos de la teoría de conjuntos (\cup, \cap, \subset, \in , etc.) y las conectivas de la lógica clásica. Por ejemplo, el razonamiento "todos los hombres son mortales, Sócrates es un hombre, luego Sócrates es mortal" se formalizaría así en lógica de clases:

$$((H \subset M) \wedge (s \in H)) \rightarrow (s \in m)$$

Puesto que a toda propiedad corresponde una clase, y los predicados monádicos representan propiedades de individuos, todo lo que pueda representarse en lógica de predicados monádicos puede también representarse en lógica de clases y estudiarse con la herramienta del álgebra de Boole. Sugerimos al lector que analice mediante diagramas de Venn los razonamientos utilizados como ejemplos en el capítulo anterior.

La equivalencia entre lógica de clases y lógica de predicados monádicos se ve claramente si consideramos que la clase C de los individuos que comparten la propiedad P puede definirse, utilizando la terminología de conjuntos, así:

$$C = \{x | P(x)\}$$

Así como la lógica de clases es equivalente a la lógica de predicados monádicos y su modelo matemático es el álgebra de clases, que es un álgebra de Boole, la lógica de relaciones es equivalente a la lógica de predicados poliádicos y su modelo matemático es el álgebra de relaciones, una extensión del álgebra de Boole elaborada por de Morgan y Pierce. Por ejemplo, la relación R entre padre e hijo será la que existe entre todos los pares ordenados (x, y) tales que el predicado $P(x, y)$ se evalúa como verdadero:

$$R = \{(x, y) | P(x, y)\}$$

En el álgebra de relaciones se definen la relación universal y la relación vacía, y un conjunto de operaciones: complementación, unión, suma, diferencia, selección, proyección y productos de relaciones. Su interés informático radica en el hecho de constituir un modelo matemático de gran utilidad para el diseño de un tipo de bases de datos llamadas, precisamente, *relacionales*.

2.3. Lógica de predicados con identidad

Muchos enunciados establecen la relación de identidad entre individuos. Por ejemplo, en

"España es el más meridional de los países europeos"

establecemos una identidad entre "España" y "el más meridional de los países europeos". Este uso del verbo "ser" para denotar la identidad es muy distinto de otros que hemos visto antes, como el "ser" de la predicación ("España es un país europeo"), el "ser" de la pertenencia ("España pertenece a la clase de los países europeos") o el "ser" de la inclusión ("la clase de los españoles está incluida en la clase de los europeos").

En principio, la identidad no es más que un predicado diádico como cualquier otro. Si los lógicos le dan una importancia especial es porque muchos razonamientos sólo pueden explicarse ampliando el lenguaje de la lógica de predicados para que lo considere como un caso especial. Por ejemplo:

P1: El que inventó la palabra "telemática" es abulense

P2: El que inventó la palabra "telemática" ha escrito "La vida en un chip"

P3: El que ha escrito "La vida en un chip" es Luis Arroyo

C: Luis Arroyo es abulense

Obsérvese que en *P1* y en *C* se utiliza el "ser" de la predicación (o de la pertenencia, según se adopte la óptica de la lógica de predicados o de la lógica de clases), mientras que en *P2* y *P3* es el "ser" de la identidad.

Tradicionalmente, en lógica se definen las "*descripciones*" para tratar con enunciados de ese tipo, y se introducen un nuevo símbolo, =, para la relación de igualdad, y un nuevo cuantificador, ι (iota), para formalizar expresiones del tipo "el x tal que...", de modo que el razonamiento anterior se formaliza así:

$$P1: A ((\iota x) (I(x, t)))$$

$$P2: (\iota x) (I(x, t)) = (\iota x) (E(x, v))$$

$$P3: (\iota x) (E(x, v)) = a$$

$$C: A(a)$$

con $i(A)$ = "abulense", $i(I)$ = "inventor de", $i(E)$ = "autor de",
 $i(t)$ = "la palabra telemática", $i(v)$ = "la vida en un chip", $i(a)$ = "Luis Arroyo".

Y una ampliación del sistema axiomático permite analizar razonamientos de ese tipo.

Pero tenemos otro recurso expresivo para formalizar ese tipo de razonamientos: las funciones (capítulo 4, apartado 1.8). Introduciendo las funciones $f_i(y)$ y $f_e(y)$ en sustitución de "el x tal que x inventó y " y "el x tal que x ha escrito y ", podemos escribir:

$$P1: A(f_i(t))$$

$$P2: f_i(t) = f_e(v)$$

$$P3: f_e(v) = a$$

$$C: A(a)$$

Nuestro sistema inferencial sólo necesita de la adición de una regla mediante la cual pueda sustituirse cualquier término por otro que sea idéntico a él.

2.4. Lógica modal

Una variable proposicional p (o, en su caso, una fórmula atómica) expresa "es el caso que p ", o " p es verdadera", mientras que $\neg p$ expresa "no es el caso que p ", o " p es falsa". La lógica modal se introdujo para poder dar cuenta de las llamadas "expresiones modales" o "modalidades", que son las que incluyen declaraciones del tipo "es necesario que", "es posible que", "es imposible que".

En lógica modal se definen dos nuevas conectivas unarias u *operadores modales*, uno para la posibilidad, \Diamond y otro para la necesidad, \Box . "Es posible que p " se representa por " $\Diamond p$ ", y "es necesario que p ", por " $\Box p$ ". En realidad, bastaría con uno solo para expresar las tres modalidades enunciadas:

"es posible que p ": $\Diamond p$, o bien $\neg \Box \neg p$

"es necesario que p ": $\Box p$, o bien $\neg \Diamond \neg p$

"es imposible que p ": $\neg \Diamond p$, o bien $\Box \neg p$

Mediante estos nuevos recursos expresivos, los lógicos han formalizado de manera más convincente la relación de implicación. A diferencia de la "implicación material" ($A \rightarrow B$, recuérdese la discusión sobre el significado del condicional en el apartado 1.3 del capítulo 2) y de la "implicación lógica" ($A \models B$, apartado 5.2 del capítulo 2), la "implicación estricta", $A \Rightarrow B$, se define así:

$$\neg \Diamond (A \wedge \neg B)$$

("no es posible que, siendo A verdadera, B sea falsa"), o, equivalentemente:

$$\Box (\neg A \vee B)$$

("necesariamente, o bien A es falsa, o bien B verdadera, o ambas cosas").

El uso de esta noción requiere, a efectos semánticos, el concepto de "mundos posibles": $\Diamond A$ será verdadera si A lo es en alguno de los mundos posibles, y $\Box A$ será verdadera si A lo es en todos los mundos posibles.

En informática, la lógica modal tiene aplicaciones en la teoría de lenguajes de programación y en los sistemas inferenciales llamados "no monótonos", que son aquellos en los que se obtienen conclusiones provisionales que pueden verse invalidadas a la luz de nuevas evidencias. En la *lógica no monótona* se añade otro operador modal, " M ", que significa "es consistente con lo que se sabe hasta ahora". Por ejemplo, consideremos la sentencia:

$$(\forall x) (\forall y) (P(x, y) \wedge M \neg D(x) \rightarrow A(x, y))$$

cuya interpretación puede ser: "si x es padre de y , a menos que x sea un padre desnaturalizado, entonces x ama a y ". Si previamente se ha demostrado (por medio de otras sentencias, o porque se da como un hecho) $P(a, b)$ pero no $D(a)$, entonces la sentencia nos permite inferir provisionalmente $A(a, b)$.

Además, la lógica no monótona permite formalizar un tipo de razonamiento no deductivo muy interesante que se llama "*abducción*"². Para ilustrarlo con un ejemplo muy sencillo, supongamos que tenemos la sentencia:

$$(\forall x) (G(x) \rightarrow F(x))$$

que representa el elemento de conocimiento "todos los pacientes con gripe presentan fiebre". Sabemos que, dado $G(a)$, la sentencia permite inferir, gracias al *modus ponens*, $F(a)$. Pero ¿puede procederse al revés? Es decir, si sabemos que $F(a)$, ¿podemos *deducir* $G(a)$? La respuesta, desde luego, es "no": el razonamiento

$$\frac{(\forall x) (G(x) \rightarrow F(x)) \quad F(a)}{G(a)}$$

no es un razonamiento válido (apartado 5.3 del capítulo 2). Ahora bien, la mente humana funciona a veces de un modo que podemos asimilar a este tipo de razonamiento: "si veo que este paciente presenta fiebre, puedo suponer, en principio, que tiene gripe, a menos que haya descartado esta hipótesis por otro motivo". Este es un ejemplo de *inferencia abductiva*. Con los operadores modales podemos formalizarlo así:

² Aclaremos, sólo para los lectores interesados en los "ovnis", que este término no tiene nada que ver aquí con actividades de seres extraterrestres.

$$\begin{array}{c}
 (\forall x) (G(x) \rightarrow F(x)) \\
 F(a) \\
 \hline
 \Diamond G(a)
 \end{array}$$

Naturalmente, puede haber otras hipótesis consistentes con el conocimiento previo, además de "G". Al mecanizar este tipo de razonamiento tenemos que establecer una medida de credibilidad para las distintas hipótesis. Veremos un enfoque basado en probabilidades en el apartado 3.2 del siguiente capítulo.

2.5. Lógica temporal

Considerada por algunos autores como un tipo de lógica modal, la lógica temporal es aquella en la que la función de evaluación depende del instante. Se introducen para ello dos predicados temporales:

$F(A)$: A será verdadera en algún instante futuro
 $P(A)$: A fue verdadera en algún instante pasado

a partir de los cuales pueden definirse otros, por ejemplo:

$\neg F(\neg A)$: A será verdadera en todo instante futuro
 $\neg P(\neg A)$: A fue verdadera en todo instante pasado

y un predicado de precedencia temporal, $T(t_1, t_2)$, que será verdadero si $t_1 < t_2$ y falso en caso contrario, y con el cual se puede extender la función de evaluación a los nuevos predicados:

$E(t, F(A))=1$ si y sólo si hay un t' tal que $T(t, t')$ y $E(t', A)=1$
 $E(t, P(A))=1$ si y sólo si hay un t' tal que $T(t', t)$ y $E(t', A)=1$

En informática, la lógica temporal tiene aplicaciones para los trabajos sobre especificación y verificación de programas, así como para todos aquellos en los que interviene la concurrencia y la intercomunicación entre procesos. Asimismo, permite formalizar razonamientos basados en conocimiento de tipo dinámico (por ejemplo, reglas en las que la satisfacción de sus componentes incluye la modalidad temporal).

3. Lógicas multivaloradas

La idea que motiva a todas las lógicas multivaloradas, también llamadas multivalentes o polivalentes, radica en el hecho de que a veces somos incapaces

de asignar valores de "verdad" o "falsedad" absolutos a las sentencias. Ahora bien, esto es algo que ya habíamos previsto (definición 3.1.2 del capítulo 2): decíamos que el conjunto de valores de verdad, V , debe tener, *como mínimo*, dos valores. Hasta ahora, hemos considerado que $\text{card}(V)=2$. Las lógicas multivaloradas son aquellas en las que $\text{card}(V)>2$.

Consideremos el caso más sencillo: $\text{card}(V) = 3$. Adoptemos los símbolos "1" para "verdadero", "0" para "falso" y " $1/2$ " para "ni verdadero ni falso". En esta *lógica trivalorada* (o *trivalente*) la extensión del dominio de la función de evaluación para las sentencias puede hacerse teniendo en cuenta la siguiente tabla que define las conectivas en el conjunto V :

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
0	0	1	0	0	1	1
0	$1/2$	1	0	$1/2$	1	$1/2$
0	1	1	0	1	1	0
$1/2$	0	$1/2$	0	$1/2$	$1/2$	$1/2$
$1/2$	$1/2$	$1/2$	$1/2$	$1/2$	1	1
$1/2$	1	$1/2$	$1/2$	1	1	$1/2$
1	0	0	0	1	0	0
1	$1/2$	0	$1/2$	1	$1/2$	$1/2$
1	1	0	1	1	1	1

(Como puede comprobarse, si se eliminan las filas en las que A o B se evalúan como " $1/2$ " queda la tabla binaria clásica).

Para la evaluación de sentencias con variables cuantificadas bastará tener en cuenta que " \forall " y " \exists " son generalizaciones de " \wedge " y " \vee " respectivamente:

$$E[(\forall x)(A(x))] = 1 \text{ si } (\forall x)(E(A(x)) = 1)$$

$$= 0 \text{ si } (\exists x)(E(A(x)) = 0)$$

$$= 1/2 \text{ en otro caso}$$

$$E[(\exists x)(A(x))] = 1 \text{ si } (\exists x)(E(A(x)) = 1)$$

$$= 0 \text{ si } (\forall x)(E(A(x)) = 0)$$

$$= 1/2 \text{ en otro caso}$$

Invitamos al lector a reflexionar sobre la justificación de estos significados de las conectivas, considerando interpretaciones concretas. Por ejemplo, el condicional puede analizarlo sobre la interpretación "si corro, entonces me canso" que considerábamos en el capítulo 2 para ilustrar el sentido del condicional en el caso de evaluación binaria.

La generalización para cualquier V tal que $\text{card}(V) > 3$ puede hacerse mediante las siguientes definiciones (llamadas *leyes de Lukasiewicz*):

$$E(\neg A) = 1 - E(A)$$

$$E(A \wedge B) = \min(E(A), E(B))$$

$$E(A \vee B) = \max(E(A), E(B))$$

$$E(A \rightarrow B) = 1 \text{ si } E(A) \leq E(B)$$

$$= 1 - E(A) + E(B) \text{ si } E(A) > E(B)$$

$$E((\forall x)(A(x))) = \min_x(E(A(x)))$$

$$E((\exists x)(A(x))) = \max_x(E(A(x)))$$

que, como puede comprobarse fácilmente, coinciden con las dadas por la tabla para el caso particular de la lógica trivalorada ($\text{card}(V)=3$).

Si hacemos $V = \{x | 0 \leq x \leq 1\}$ tendremos una lógica con infinitos valores de evaluación, que puede ponerse en relación con la *lógica probabilística*, en la que las conectivas se corresponden con operaciones de la teoría de probabilidades. Para el operador de negación, esta correspondencia es inmediata; así, por ejemplo, si tiramos un dado, en el *lenguaje de las probabilidades* decimos: "el suceso 'sacar un tres' tiene probabilidad $1/6$, y el suceso contrario tiene probabilidad $1-1/6 = 5/6$ ", mientras que en el *lenguaje de la lógica* diríamos: "la sentencia 'al tirar un dado sale un tres' tiene el valor de verdad $1/6$, mientras que la sentencia 'al tirar un dado sale un número distinto de tres' tiene el valor de verdad $1-1/6 = 5/6$ ".

En una lógica multivalorada podemos definir la equivalencia entre sentencias del mismo modo que en la lógica binaria (apartado 3.5 del capítulo 2). Es decir, dos sentencias son equivalentes si sus evaluaciones son las mismas para todas las interpretaciones posibles. El lector puede comprobar que la mayoría de las leyes de la lógica binaria (distributividad, asociatividad, de Morgan, etc.) se siguen cumpliendo si se adoptan las definiciones dadas para las conectivas. Por ejemplo:

$$E(\neg(A \wedge B)) = 1 - \min(E(A), E(B))$$

$$E(\neg A \vee \neg B) = 1 - \max(1-E(A), 1-E(B))$$

Si $E(A) > E(B)$ entonces resulta: $E(\neg(A \wedge B)) = E(\neg A \vee \neg B) = 1 - E(B)$, mientras que si $E(A) \leq E(B)$ resulta: $E(\neg(A \wedge B)) = E(\neg A \vee \neg B) = 1 - E(A)$. Por consiguiente,

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

Pero hay una ley importante que no se cumple: la del *tercio excluso*: $\vdash (A \vee \neg A)$, o, equivalentemente, $\vdash (\neg(A \wedge \neg A))$. En efecto:

$E(A \vee \neg A) = \max(E(A), 1 - E(A)) \geq 1/2$ (en general, distinto de 1) (en general, distinto de "1")

$E(A \wedge \neg A) = \min(E(A), 1 - E(A)) \leq 1/2$ (en general, distinto de 0) (en general, distinto de "0")

El modelo algebraico es un "*álgebra blanda*": un álgebra de Boole desposeída del axioma A3 enunciado en el apartado 4.1 del capítulo 2.

4. Lógica borrosa

4.1. Justificación

En el siguiente capítulo presentaremos los principios de la aplicación de la lógica a los llamados "sistemas expertos". Como allí veremos, un problema fundamental a considerar es el de la representación en la máquina de los conocimientos y los procedimientos inciertos e imprecisos que utilizan los expertos humanos para resolver problemas. En la mayoría de los sistemas expertos que se han construido o se están diseñando se adoptan técnicas "ad hoc" para atacar ese problema. Pero existen intentos teóricos para introducir en la lógica formal la imprecisión y la subjetividad propia de la actividad humana, y parece obvio que en el futuro se tienda a basar rigurosamente el diseño de los sistemas expertos en tales teorías. La más conocida de ellas es la lógica borrosa ("fuzzy logic"). Las consideraciones anteriores, unidas a la escasez de bibliografía en español sobre el tema, nos han incitado a dedicar una atención especial a la lógica borrosa en este capítulo de "otras lógicas".

La lógica borrosa va aún más allá que las lógicas con infinitos valores de verdad. Porque ya no sólo se trata de considerar que hay una infinidad de valores semánticos entre "verdadero" y "falso", sino también de tener en cuenta que estos mismos valores de verdad son imprecisos. Por ejemplo, a la sentencia "este párrafo es de difícil comprensión" podría asignársele el valor de verdad 0,7 en lógica multivalorada. Pero generalmente hacemos inferencias imprecisas, como "si alguien encuentra muy difícil comprender un párrafo, casi con seguridad abandona la lectura; este párrafo es de difícil comprensión para tal persona, luego

es probable que esa persona abandone la lectura". La lógica multivalorada no nos permite hacer inferencias de ese tipo, porque intervienen en ella matices (relaciones entre "difícil" y "muy difícil", entre "casi con seguridad" y "es probable") imposibles de abordar con la simple extensión del conjunto de valores de verdad.

Ese tipo de inferencia imprecisa es el que aborda la lógica borrosa. Y para ello parte de una reconsideración del mismo pilar básico de las matemáticas: el concepto de *conjunto*. En la realidad se presentan situaciones (particularmente, cuando aparecen consideraciones subjetivas) en las que resulta difícil determinar la pertenencia o no de un elemento a un conjunto. Por ejemplo:

- el conjunto de los números naturales mucho mayores que 100: parece claro que 101 no pertenece al conjunto, y que 10^{10} sí, pero ¿y 500?;
- el conjunto de las personas pobres: ¿pertenezco yo a ese conjunto?;
- el conjunto de las mujeres preciosas, etc.

Tales conjuntos pueden denominarse "borrosos" (o "difusos") para indicar que no existe un criterio que determine exactamente un límite entre pertenencia o no pertenencia al conjunto.

Pero, en este momento, el lector atento puede objetar: "lo que ocurre es que no se ha establecido el criterio para definir los conjuntos. Así, en el primer ejemplo, puede decirse (por convenio, o por hipótesis de trabajo) que el límite está en 1000; en el segundo, que es pobre toda persona que, sin tener patrimonio, perciba unos ingresos inferiores al salario mínimo; y en el último ejemplo ya es más difícil establecer un criterio, porque ¿a quién se le ocurre tratar de determinar matemáticamente tal conjunto?".

Ahora bien, si tratamos de formalizar las relaciones del hombre con su entorno siempre vamos a encontrarnos con elementos imprecisos o "borrosos", sobre todo en la actividad más típicamente humana, la comunicación por medio del lenguaje. Cuando alguien está aprendiendo a conducir, en un determinado momento puede recibir una orden del instructor como ésta: "levante ligera y lentamente el pie del embrague", pero nunca una como ésta: "levante el pie 8° a una velocidad de $2^\circ 30'$ por segundo". Y sin embargo el hombre, considerado como sistema, se comporta bien (aprende) con entradas "borrosas" como las del primer caso, mientras sería difícil que lo hiciera con las del segundo tipo.

A poco que se reflexione, se llegará a la conclusión de que si pretendemos analizar sistemas muy complejos como el hombre, las sociedades, etc. (ya sea con espíritu puramente científico, ya sea con fines utilitarios: diseño de máquinas que puedan razonar, tomar decisiones, comprender el lenguaje natural, etc.), resulta imprescindible introducir en los modelos la imprecisión y la subjetividad propias de la actividad humana. Además, en estos sistemas complejos hay que tener en cuenta el *principio de incompatibilidad de Zadeh*:

"A medida que aumenta la complejidad de un sistema, nuestra capacidad para hacer afirmaciones sobre su comportamiento que sean precisas y, al mismo tiempo, significativas, va disminuyendo, hasta alcanzar un umbral por debajo del cual precisión y significación (o pertinencia) llegan a ser características casi mutuamente excluyentes".

Como consecuencia, los resultados muy precisos (hacia los que se orientan las herramientas matemáticas clásicas aplicadas en ingeniería) tienen poca utilidad por su incapacidad para representar significativamente el comportamiento de un sistema complejo. De esta forma, resultan más interesantes los modelos cualitativos.³

Estas consideraciones conducen a una reformulación del concepto básico de conjunto, admitiendo grados de pertenencia de los elementos a los conjuntos. Puesto que la lógica borrosa se apoya en la teoría de conjuntos borrosos, será preciso que veamos previamente los elementos básicos de esa teoría.

4.2. Subconjuntos borrosos

En la teoría clásica, dado un elemento x de un universo U , y un subconjunto, $A \subset U$, hay dos posibilidades: $x \in A$ o $x \notin A$. Puede definirse una función característica de pertenencia, μ_A , tal que $\mu_A(x) = 1$ si $x \in A$ y $\mu_A(x) = 0$ si $x \notin A$. Es fácil demostrar que:

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x)$$

$$\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x)$$

$$\mu_{A \cup B}(x) = \mu_A(x) + \mu_B(x) \text{ (suma lógica)}$$

Definición 4.2.1. Dado un universo U , un subconjunto borroso, A , de U , es un conjunto de pares

$$\{ (x | \mu_A(x)) \}, \forall x \in U,$$

donde $\mu_A(x)$ es una función que toma sus valores en un conjunto M llamado *conjunto de pertenencia*.

³ Los conjuntos y la lógica borrosa que vamos a estudiar aquí son una aproximación a la modelación de estos sistemas complejos, pero no la única. En las últimas décadas se han propuesto y desarrollado al menos otras dos: la simulación cualitativa y la teoría de las catástrofes. Creemos que estas aproximaciones convergerán, finalmente, en una nueva *matemática cualitativa*.

Generalmente se toma $M = \{m | m \in [0, 1]\}$; si se hace $M = \{0, 1\}$, entonces A se reduce a un subconjunto ordinario, de manera que la teoría clásica de conjuntos es un caso particular de la teoría de conjuntos borrosos.

Ejemplo 4.2.2. Si $U = \{1, 2, 3, \dots, 10\}$, podemos definir el conjunto "varios", V , como:

$$V = \{3|0, 5; 4|0, 8; 5|1; 6|1; 7|0, 8; 8|0, 5\}$$

La asignación de valores a $\mu_V(x)$ es totalmente subjetiva, de manera que otra persona daría, con toda probabilidad, otras cifras.

Ejemplo 4.2.3. Si $U = \{x | x \in [0, 150]\}$ se interpreta como el conjunto de edades posibles de un ser humano, podrían definirse los subconjuntos borrosos J (joven) y V (viejo) así:

$$J = \{(x|1)_{0 \leq x \leq 40}; (x|(1 + (x-40)^2/40)^{-1})_{x > 40}\}$$

$$V = \{(x|0)_{0 \leq x \leq 40}; (x|(1 + 40/(x-40)^2)^{-1})_{x > 40}\}$$

Podemos tener una visión gráfica de estos conjuntos representando $\mu_J(x)$ y $\mu_V(x)$ (figura 5.1):

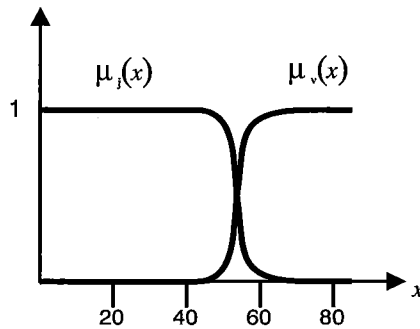


Figura 5.1.

Definición 4.2.4. Se definen las relaciones de igualdad e inclusión entre subconjuntos borrosos del siguiente modo:

Igualdad: $A = B$ si $\mu_A(x) = \mu_B(x), \forall x \in U$

Inclusión: $A \subset B$ si $\mu_A(x) \leq \mu_B(x), \forall x \in U$

Definición 4.2.5. Se definen las operaciones de complementación, intersección, unión, producto y potenciación del siguiente modo:

Complementación: $\left(\begin{array}{l} \tilde{A} = \tilde{B} \text{ si } \mu_{\tilde{A}}(x) = 1 - \mu_{\tilde{B}}(x), \forall x \in U \\ (\text{supuesto que } M = [0,1]) \end{array} \right)$

Intersección: $\tilde{C} = \tilde{A} \cap \tilde{B} \text{ si } \mu_{\tilde{C}}(x) = \min(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x)), \forall x \in U$

Unión: $\tilde{C} = \tilde{A} \cup \tilde{B} \text{ si } \mu_{\tilde{C}}(x) = \max(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x)), \forall x \in U$

Producto: $\tilde{C} = \tilde{A} \tilde{B} \text{ si } \mu_{\tilde{C}}(x) = \mu_{\tilde{A}}(x) \cdot \mu_{\tilde{B}}(x), \forall x \in U$

Potenciación: $\tilde{C} = \tilde{A}^{\alpha} \text{ si } \mu_{\tilde{C}}(x) = \mu_{\tilde{A}}^{\alpha}(x), \forall x \in U$

Es fácil comprobar que las tres primeras se reducen a las definiciones clásicas en conjuntos ordinarios para $M = \{0, 1\}$. Asimismo, que todas las propiedades de estas operaciones en los conjuntos ordinarios (asociatividad, distributividad, etc.) se siguen cumpliendo, salvo en lo que respecta a dos muy importantes:

$$\tilde{A} \cap \tilde{\tilde{A}} \neq \Phi$$

(el conjunto vacío, Φ , es aquel subconjunto borroso de U tal que $(\forall x \in U) (\mu_{\Phi}(x) = 0)$), y

$$\tilde{A} \cup \tilde{\tilde{A}} \neq U$$

Ejemplo 4.2.6. El complemento de "varios" tal como se definió más arriba sería:

$$\tilde{\tilde{V}} = \{1|1; 2|1; 3|0,5; 4|0,2; 7|0,2; 8|0,5; 9|1; 10|1\}$$

Y las representaciones gráficas de \tilde{V} y $\tilde{\tilde{V}}$ serían las de la figura 5.2.

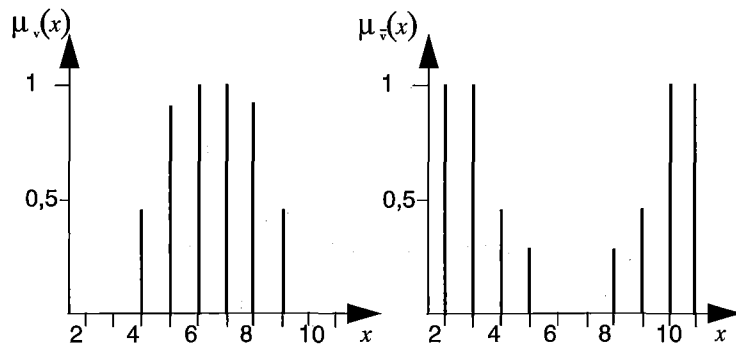


Figura 5.2.

Ejemplo 4.2.7. Con las definiciones dadas para "joven" y "viejo" en el Ejemplo 4.2.3, en la figura 5.3 pueden verse las representaciones gráficas de la intersección y la unión de ambos conjuntos.

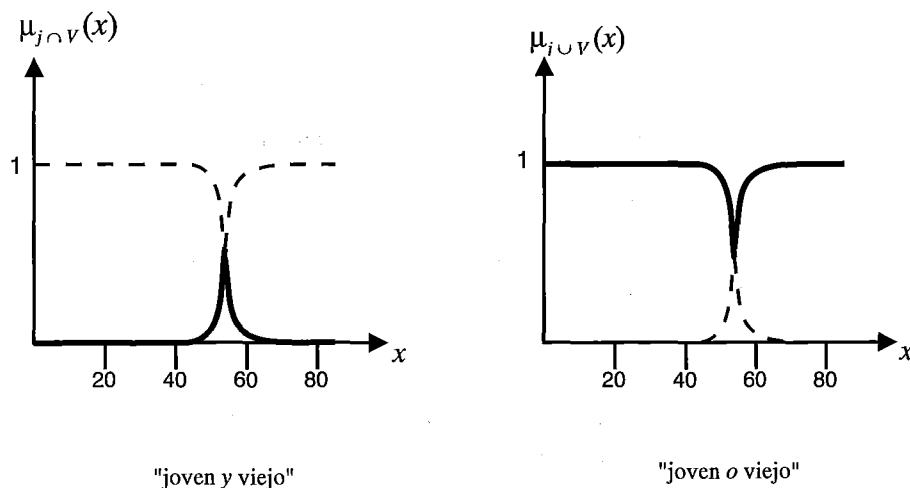


Figura 5.3.

En este ejemplo, tal como se definieron los subconjuntos, se cumple que $V = \bar{J}$, porque $(\forall x) (\mu_J(x) + \mu_V(x)) = 1$.

4.3. Relaciones borrosas

4.3.1. Relaciones ordinarias entre conjuntos borrosos

Más arriba hemos definido las relaciones de igualdad e inclusión entre conjuntos borrosos. Tales conjuntos son, realmente, subconjuntos borrosos de un universo (no borroso) U . Se trata, por tanto, de relaciones definidas sobre el conjunto de las partes (borrosas) de U , $P(U)$.

Ahora bien, aunque las partes o subconjuntos sean borrosos, las relaciones así definidas *no* son borrosas: dados $A, B \subset U$, o bien $(A, B) \in R$, o bien $(A, B) \notin R$; concretamente, para las relaciones R de igualdad e inclusión definidas, dados dos subconjuntos borrosos, A y B , o bien $A = B$, o bien $A \neq B$, y, del mismo modo, o bien $A \subset B$, o bien $A \not\subset B$.

4.3.2. Relaciones borrosas entre conjuntos ordinarios

Vamos a definir ahora el concepto de *relación borrosa* entre conjuntos *no borrosos*. Sean A y B dos subconjuntos ordinarios, en general de universos

diferentes: $A \subset U, B \subset V$, y sean $a \in A$ y $b \in B$ elementos genéricos de cada uno. Una relación borrosa entre A y B se define como un conjunto de pares ordenados (a, b) , cada uno con un determinado grado de pertenencia, μ_R , a la relación R :

$$R = \{ (a, b) | \mu_R(a, b) \}, a \in A, b \in B, 0 \leq \mu_R \leq 1 \quad [1]$$

μ_R indica así en qué grado, o con qué intensidad, los elementos a y b están en la relación R .

Por ejemplo, existe una relación entre la estación del año en que nos encontramos y el calor o frío que se siente. Esta relación es subjetiva (y, por lo tanto, borrosa), y una determinada persona podría expresarla explícitamente así:

$$\begin{aligned} U &= \{\text{estaciones}\} \\ V &= \{\text{sensaciones}\} \\ A = U &= \{\text{primavera, verano, otoño, invierno}\} \\ B &= \{\text{calor, frío}\} \end{aligned}$$

$$\begin{array}{c} \begin{array}{c} B \\ \hline c \quad f \end{array} \\ \left\{ \begin{array}{c} p \\ v \\ o \\ i \end{array} \right\} \begin{bmatrix} 0,7 & 0,4 \\ 1 & 0 \\ 0,6 & 0,5 \\ 0,1 & 1 \end{bmatrix} \end{array}$$

Obsérvese que la relación propiamente dicha se escribe con mayor comodidad en forma matricial, en lugar de hacerlo así:

$$R = \{ (p, c) | 0,7; (p, f) | 0,4; \text{etc.} \}$$

Las relaciones así definidas son binarias; la generalización a órdenes superiores (relaciones entre más de dos conjuntos) es inmediata. Así, en el ejemplo anterior un tercer conjunto podría ser el de países; su representación matricial se haría escribiendo varias matrices.

Siguiendo con la relación binaria definida por [1], se observa fácilmente que R es, realmente, un subconjunto borroso del producto cartesiano de A y B :

$$R \subset A \times B$$

En efecto, como A y B son ordinarios, su producto cartesiano consta de todos los pares (a, b) , y la relación borrosa le da un grado de pertenencia a cada uno.

En este caso binario la relación se suele escribir también así: ARB .

En general una relación borrosa n -aria entre n conjuntos ordinarios, A_1, A_2, \dots, A_n (no tienen por qué ser todos distintos) será un subconjunto borroso del producto cartesiano de todos ellos:

$$R \subset A_1 \times A_2 \times \dots \times A_n$$

O, también,

$$R = \{ (a_1, a_2, \dots, a_n) \mid \mu_R(a_1, a_2, \dots, a_n) \}, a_1 \in A_1, \dots, a_n \in A_n$$

4.3.3. Relaciones borrosas entre conjuntos borrosos

En lo sucesivo vamos a prescindir del símbolo " \sim " bajo los nombres de los conjuntos y de relaciones, y, salvo que digamos lo contrario, supondremos que todos los conjuntos y todas las relaciones son borrosos (y, de todos modos, ya sabemos que un conjunto ordinario es un caso particular de conjunto borroso, y una relación ordinaria lo es de relación borrosa).

Sean $A \subset U, B \subset V$, subconjuntos borrosos de universos U y V , y sean $a \in A, b \in B$. Se define el *producto cartesiano* de A y B del siguiente modo:

$$A \times B \triangleq \{ (a, b) \mid \min(\mu_A(a), \mu_B(b)) \} \quad [2]$$

(Es, pues, un subconjunto borroso de $U \times V$).

Cualquier subconjunto de $A \times B$ será una relación borrosa entre ambos: $ARB \subset A \times B$. Obsérvese que como todo subconjunto debe satisfacer a la relación de inclusión, el grado de pertenencia de (a, b) a tal subconjunto debe ser igual o inferior a $\min(\mu_A(a), \mu_B(b))$.

La generalización al caso n -ario es también inmediata.

4.3.4. Composición de relaciones

Dados tres conjuntos A, B, C , una relación binaria entre A y B :

$$AR_1B = \{ (a, b) \mid \mu_{R_1}(a, b) \},$$

y otra entre B y C :

$$BR_2C = \{ (b, c) \mid \mu_{R_2}(b, c) \},$$

se define la *relación compuesta* de R_1 y R_2 del siguiente modo:

$$A(R_1 \circ R_2)C = \{ (a, c) \mid \max_b [\min(\mu_{R_1}(a, b), \mu_{R_2}(b, c))] \} \quad [3]$$

Es decir, para cada pareja (a, c) se toma como grado de pertenencia a $R_1 \circ R_2$ el resultado de la siguiente operación:

- Para cada uno de todos los $b \in B$ se toma el mínimo de las funciones de pertenencia a las relaciones R_1 y R_2 de las parejas (a, b) y (b, c) , respectivamente.
- El máximo de todos los mínimos anteriores es el valor de función de pertenencia a $R_1 \circ R_2$ de la pareja (a, c) .

Si A, B, C , son finitos, R_1 y R_2 pueden escribirse en forma matricial, y puede comprobarse que la matriz correspondiente a $R_1 \circ R_2$ se calcula como el "producto máx-mín" de las matrices R_1 y R_2 . El producto máx-mín de dos matrices es el producto ordinario, pero sustituyendo la operación "suma" por la operación "máximo" y el "producto" por el "mínimo".

Estas definiciones, como las que daremos luego, pueden parecer a primera vista arbitrarias, y hasta cierto punto lo son. No obstante, en su elección se han tenido en cuenta los siguientes criterios:

- a) que sean consistentes con las definiciones paralelas de la teoría de conjuntos ordinarios, es decir, que ésta pueda considerarse como un caso particular de la teoría de conjuntos borrosos;
- b) que los modelos basados en la teoría reflejen razonablemente bien la realidad; y
- c) que las computaciones que se derivan de la utilización de los modelos sean sencillas y, por tanto, se ejecuten con rapidez; en particular, las operaciones de selección de máximo o mínimo requieren mucho menos tiempo que las de suma o producto.

Veamos un ejemplo para ilustrar la composición de relaciones. Sean A, B y R_1 las definidas en el ejemplo anterior (relación entre estaciones del año y sensaciones de frío o calor). Obsérvese que en ese caso particular tanto A como B se han definido como conjuntos ordinarios; en cualquier caso, R_1 es un subconjunto borroso de $A \times B$, independientemente de que A y B sean ordinarios o borrosos. Podríamos seguir con esos mismos conjuntos y definir otro, borroso o no, relacionado con B y ver la relación compuesta. Pero para aplicar tanto la definición [2] como la [3], vamos a redefinir A y B como conjuntos borrosos:

$$U = \{p, v, o, i\},$$

donde p = primavera; v = verano; o = otoño; i = invierno.

$$V = \{T_1, T_2\},$$

donde T_1 = temperatura superior a 20° C; T_2 = temperatura igual o inferior a 20° C (universos no borrosos).

$$A = \text{estaciones frías} = \{p|0,3; v|0,1; o|0,4; i|0,9\}$$

$$B = \text{sensación de frío} = \{T_1|0,2; T_2|0,8\}$$

El producto cartesiano de A y B , según [2], expresado en forma matricial será:

$$A \times B = \begin{matrix} & \begin{matrix} T_1 & T_2 \end{matrix} \\ \begin{matrix} p \\ v \\ o \\ i \end{matrix} & \begin{bmatrix} 0,2 & 0,3 \\ 0,1 & 0,1 \\ 0,2 & 0,4 \\ 0,2 & 0,8 \end{bmatrix} \end{matrix}$$

Una relación entre A y B es, según hemos dicho, cualquier subconjunto de $A \times B$. Por ejemplo:

$$R_1 = \begin{matrix} & \begin{matrix} T_1 & T_2 \end{matrix} \\ \begin{matrix} p \\ v \\ o \\ i \end{matrix} & \begin{bmatrix} 0,2 & 0,2 \\ 0,1 & 0 \\ 0,2 & 0,3 \\ 0 & 0,8 \end{bmatrix} \end{matrix}$$

Obsérvese que R_1 relaciona bastante bien la sensación de frío alta con las estaciones, pero no tanto la sensación de poco frío con las mismas (particularmente, el grado de pertenencia de (v, T_1) a R_1 debería ser mucho más alto del 0,1 que resulta).

La explicación es que, tal como se han definido A y B , R_1 relaciona la sensación de *frío* con las estaciones *frías*, pero no la sensación de poco frío con las estaciones poco frías. Para que así fuera, deberíamos definir una nueva relación haciendo uso de las operaciones de unión y complementación. Volveremos sobre este ejemplo en el apartado 4.4.2.

Consideremos ahora un nuevo subconjunto borroso, C , en el universo $W = \{\text{ropas de abrigo}\}$, establecido así:

$$C = \{b|0,1; t|0,5; a|0,9\}$$

donde b = bañador; t = traje; a = abrigo.

El producto cartesiano de B y C es:

$$B \times C = \begin{matrix} & b & t & a \\ \begin{matrix} T_1 \\ T_2 \end{matrix} & \begin{bmatrix} 0,1 & 0,2 & 0,2 \\ 0,1 & 0,5 & 0,8 \end{bmatrix} \end{matrix}$$

Tomemos $R_2 = B \times C$ (podría ser cualquier otro subconjunto).

Definidas así AR_1B y BR_2C , A y C estarán en la relación $R_1 \circ R_2$, que se calcula según [3], haciendo el producto máx-mín de las matrices de R_1 y R_2 :

$$\begin{matrix} & \begin{matrix} T_1 & T_2 \end{matrix} \\ \begin{matrix} p \\ v \\ o \\ i \end{matrix} & \begin{bmatrix} 0,2 & 0,2 \\ 0,1 & 0 \\ 0,2 & 0,3 \\ 0 & 0,8 \end{bmatrix} \end{matrix} \cdot \begin{matrix} & b & t & a \\ \begin{matrix} T_1 \\ T_2 \end{matrix} & \begin{bmatrix} 0,1 & 0,2 & 0,2 \\ 0,1 & 0,5 & 0,8 \end{bmatrix} \end{matrix} = \begin{matrix} & b & t & a \\ \begin{matrix} p \\ v \\ o \\ i \end{matrix} & \begin{bmatrix} 0,1 & 0,2 & 0,2 \\ 0,1 & 0,1 & 0,1 \\ 0,1 & 0,3 & 0,3 \\ 0,1 & 0,5 & 0,8 \end{bmatrix} \end{matrix}$$

Al interpretar esta relación entre "prendas de abrigo" y "estaciones frías" hemos de hacer el mismo comentario que antes con respecto a R_1 .

4.4. Lógicas borrosas

4.4.1. Sintaxis y semánticas

Como sintaxis de la lógica borrosa es válida la misma definida para la lógica de predicados (aunque en lo que sigue prescindiremos de las funciones, para simplificar). Pero hay varias maneras de definir la semántica, que conducen a varias maneras de entender la lógica borrosa (de ahí el plural en el título de este apartado).

Empecemos por el sentido más restringido, en el que la lógica borrosa es un tipo de lógica multivalorada. La semántica viene definida por una función de interpretación que asigna a cada predicado monádico un conjunto borroso, y a cada predicado poliádico una relación borrosa (entre conjuntos ordinarios o borrosos). La evaluación de un predicado es la función de pertenencia del conjunto o relación que corresponda a ese predicado. Para evaluar sentencias (es decir, para extender el dominio de la función de evaluación) se suelen aplicar las leyes de Lukasiewicz enunciadas más arriba (en el apartado 3), aunque para el condicional se utiliza más esta otra definición:

$$E(A \rightarrow B) = \max(\min(E(A), E(B)), 1 - E(A))$$

El lector puede comprobar que esto es lo mismo que convenir en que la sentencia $A \rightarrow B$ es equivalente a $\neg A \vee (A \wedge B)$ (observe también que esta última sentencia es equivalente a $\neg A \vee B$ en lógica binaria, pero no en una lógica multivalorada).

Pero hay una concepción más amplia de la lógica borrosa: independientemente de que los predicados puedan ser borrosos o no, los valores de "verdad" o "falsedad" pueden ellos mismos ser borrosos. En este marco nos situaremos de ahora en adelante. Pero antes de seguir, ¿podría construirse un sistema axiomático? La respuesta es "no" (sin "borrosidad"). En efecto, consideremos un simple predicado, como "viejo". Aplicado a un individuo concreto, tiene que tener una evaluación. En una lógica multivalorada como las vistas hasta ahora podríamos decir, por ejemplo, sabiendo que "Juan" tiene cincuenta años, y conviniendo que el conjunto borroso correspondiente a viejo es el de la figura 5.1, que $E(V(\text{Juan})) = \mu_V(50) = 0,6$. Y si Juan tuviese cien años diríamos que $E(V(\text{Juan})) = 1$. Pero el nuevo marco sirve para modelar razonamientos sobre predicados imprecisos en general, no aplicados a individuos concretos. Por ejemplo: "si alguien es viejo entonces es muy poco probable que gane una maratón". La evaluación del antecedente de este condicional no es un número, sino el mismo subconjunto borroso "viejo". Es decir, como decíamos antes, los valores de "verdad" y "falsedad" son borrosos. En una lógica multivalorada puede haber sentencias cuya evaluación sea siempre 1 (verdadera), que serán sentencias válidas (tautologías, en el caso de la lógica de proposiciones multivalorada), y, por tanto, se podrá construir un sistema axiomático que permita, a partir de unos axiomas, derivar otras sentencias válidas. Pero en lógica borrosa no puede definirse el concepto de "sentencia válida", y por ello no puede haber sistema axiomático, y, por lo mismo, tampoco tiene sentido hablar de "completitud" ni de "consistencia". Esto aleja tanto a la lógica borrosa de la "tradición lógica" que muchos autores discuten que se le pueda llamar "lógica". La única defensa de la lógica borrosa (por ahora, mientras no se revise todo el cuerpo de la lógica) es pragmática: si con ella se pueden modelar situaciones y construir sistemas que no se pueden abordar con otras lógicas, y si estos modelos permiten comprender mejor las situaciones y diseñar sistemas que funcionan, tanto peor para la "tradición lógica".

El conjunto de valores de verdad en esta lógica borrosa no es, pues, el conjunto de puntos en el intervalo $[0,1]$, sino un conjunto de subconjuntos borrosos de ese conjunto. Esos subconjuntos borrosos no son todos los posibles, sino lo que se llaman *valores de verdad lingüísticos*. Concretamente,

$V = \{\text{verdadero, falso, no verdadero, no falso, bastante verdadero, bastante falso, poco verdadero, muy verdadero, más o menos verdadero, más bien verdadero, ...}\}$

Para cada predicado, se definirá el subconjunto correspondiente a "verdadero" (por ello, se dice que los valores de verdad en lógica borrosa son "locales").

Así, en el caso de la sentencia " x es viejo", el subconjunto correspondiente a que esta sentencia sea verdadera puede ser el definido en los ejemplos anteriores. Este subconjunto es el *significado* del predicado. Definido este subconjunto, los correspondientes a los otros valores de verdad lingüísticos pueden calcularse en función de él definiendo previamente unos convenios para "falso" y para las partículas lingüísticas "muy", "bastante", etc.:

- "falso" es el subconjunto tal que $\mu_{falso}(x) = \mu_{verdadero}(c(x))$, donde $c(x)$ es alguna función complementaria definida sobre U (por ejemplo, si $U = [1,150]$, $c(x) = (150 - x)$);
- "no verdadero" = (" $\overline{\text{verdadero}}$ ");
- "muy verdadero" = " verdadero "²
- "algo verdadero" = " verdadero "^{1/2}; etc.

Por ejemplo, en el caso de que la sentencia sea el predicado "viejo" aplicado a alguien, tendremos para "muy viejo" el significado:

$$S(\text{muy } v) = S^2(v) = \{x | \mu_{S^2(v)}(x)\},$$

con $\mu_{S^2(v)}(x) = \mu_{S(v)}^2(x)$.

En la figura 5.4 puede verse la representación gráfica de este convenio para "muy", que, más o menos, corresponde a la interpretación intuitiva de la partícula.

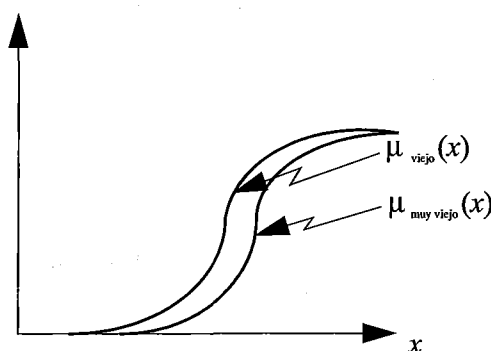


Figura 5.4.

4.4.2. Interpretación de sentencias

Para toda sentencia se tiene que poder calcular un *significado*, es decir, un subconjunto borroso que corresponde a la interpretación "verdadera" de tal sentencia. Empecemos por los casos más sencillos: los de las sentencias cons-

truidas con la negación, la conjunción y la disyunción. El cálculo del significado de tales sentencias se hace del siguiente modo:

$$S(\neg A) = \overline{S(A)}$$

$$S(A \wedge B) = S(A) \cap S(B)$$

$$S(A \vee B) = S(A) \cup S(B)$$

Como vemos, basta con utilizar las operaciones de complementación, intersección y unión entre conjuntos definidas en el apartado 4.2.

Por ejemplo, interpretemos la sentencia "Manuel es un viejo que ronda los setenta años", es decir, "Viejo (Manuel) \wedge ronda setenta (Manuel)". La interpretación (o significado) de viejo puede ser la que venimos utilizando. Hemos de dar significado a "ronda setenta años". Gráfica y aproximadamente, podría ser como indica la figura 5.5, en la que también hemos señalado la función de pertenencia al conjunto intersección de "viejo" y "ronda setenta", que es el significado de la sentencia.

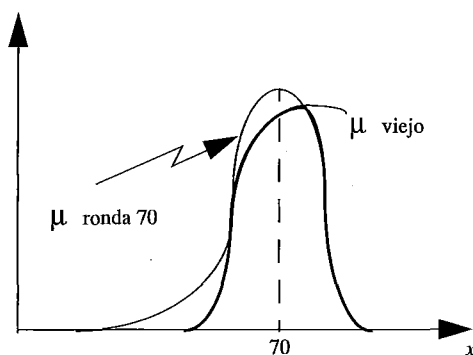


Figura 5.5.

Es fácil comprender que con esto no hacemos sino abordar los casos más sencillos. Las sentencias, normalmente, manejan palabras cuyos significados son conjuntos borrosos de universos diferentes. Por ejemplo, en la sentencia "Manuel es un viejo extraordinariamente decrepito" tenemos dos variables borrosas: edad (con valor "viejo") y estado (con valor "decrepito"). La interpretación exige el conocer la de cada una de las variables (y definir "extraordinariamente", que podría ser, por ejemplo, el operador de potenciación con valor 3), y daría un resultado bidimensional.

Veamos otro ejemplo: Con los conjuntos $A = \{\text{estaciones frías}\}$, $B = \{\text{sensación de frío}\}$ y $C = \{\text{ropas de abrigo}\}$ definidos anteriormente, tendríamos:

$$S(\text{estaciones muy frías}) = A^2 = \{p|0,09; v|0,01; o|0,16; i|0,81\};$$

$$S(\text{estaciones no muy frías}) = \overline{A^2} = \{p|0,91; v|0,99; o|0,84; i|0,19\};$$

$$S(\text{estaciones frías y no muy frías}) = A \cap \overline{A^2} = \{p|0,3; v|0,1; o|0,4; i|0,19\}$$

$$S(\text{sensación de mucho frío}) = B^2 = \{T_1|0,04; T_2|0,64\};$$

$$S(\text{sensación de no mucho frío}) = \overline{B^2} = \{T_1|0,96; T_2|0,36\};$$

$$S(\text{ropas que no abrigan}) = \overline{C} = \{b|0,9; t|0,5; a|0,1\};$$

etcétera.

Con esto, entonces, podemos interpretar sentencias que contienen negaciones, conjunciones y disyunciones. El caso del condicional necesita un tratamiento aparte.

4.4.3. Interpretación de sentencias condicionales

Consideraremos el caso general "si A , entonces B , si no, C ", es decir, " $(A \rightarrow B) \wedge (\neg A \rightarrow C)$ ", del cual " $(A \rightarrow B)$ " será un caso particular. Cuando A , B y C son sentencias ordinarias con interpretaciones binarias, el cálculo de proposiciones clásico nos permite evaluar la sentencia como verdadera o falsa. Pero cuando A , B y C son sentencias borrosas, cada una tendrá un significado, y se trata de calcular el significado de la sentencia global.

Por ejemplo:

- si la humedad de la carretera es grande, el coche tiende a patinar;
 - o
 - si estamos en una estación muy fría uso ropa de abrigo, si no, uso ropa de muy poco abrigo;
- etcétera.

Se considera que este tipo de sentencia establece una relación entre los universos U (a que pertenece $S(A)$) y V (a que pertenecen $S(B)$ y $S(C)$), relación que se define de la siguiente manera:

- $S(\text{si } A \text{ entonces } B, \text{ si no } C) = [S(A) \times S(B)] \cup [S(\overline{A}) \times S(C)]$, donde " \times " indica el producto cartesiano, definido en [2] (pág. 203).

En el caso de que C no esté especificado ("si A entonces B ") se toma $S(C) = V$ (universo de C), es decir, se considera que el consecuente de "no A " puede ser cualquier subconjunto de V .

Ejemplos:

- a) "Si la estación es fría se siente frío; si no, no se siente frío".

$$\begin{aligned}
A &= S(\text{estación fría}) = \{p|0,3; v|0,1; o|0,4; i|0,9\} \\
B &= S(\text{se siente frío}) = \{T_1|0,2; T_2|0,8\} \\
\bar{A} &= S(\text{estación no fría}) = \{p|0,7; v|0,9; o|0,6; i|0,1\} \\
\bar{B} &= S(\text{no se siente frío}) = \{T_1|0,8; T_2|0,2\}
\end{aligned}$$

La relación definida por la sentencia será:

$$\begin{array}{c}
\begin{array}{cc} T_1 & T_2 \end{array} \\
\begin{array}{c} p \\ v \\ o \\ i \end{array} \begin{bmatrix} 0,2 & 0,3 \\ 0,1 & 0,1 \\ 0,2 & 0,4 \\ 0,2 & 0,8 \end{bmatrix} \cup \begin{array}{cc} T_1 & T_2 \\ \begin{bmatrix} 0,7 & 0,2 \\ 0,8 & 0,2 \\ 0,6 & 0,2 \\ 0,1 & 0,1 \end{bmatrix}
\end{array} = \begin{array}{cc} T_1 & T_2 \\ \begin{bmatrix} 0,7 & 0,3 \\ 0,8 & 0,2 \\ 0,6 & 0,4 \\ 0,2 & 0,8 \end{bmatrix}
\end{array}
\end{array}$$

b) "Si se siente frío, se usa ropa de abrigo; si no, no".

$$\begin{aligned}
\bar{C} &= S(\text{se usa ropa de abrigo}) = \{b|0,1; t|0,5; a|0,9\} \\
C &= S(\text{no se usa ropa de abrigo}) = \{b|0,9; t|0,5; a|0,1\}
\end{aligned}$$

$$\begin{array}{c}
\begin{array}{ccc} b & t & a \end{array} \\
\begin{array}{c} T_1 \\ T_2 \end{array} \begin{bmatrix} 0,1 & 0,2 & 0,2 \\ 0,1 & 0,5 & 0,8 \end{bmatrix} \cup \begin{array}{ccc} b & t & a \\ \begin{bmatrix} 0,8 & 0,5 & 0,1 \\ 0,2 & 0,2 & 0,1 \end{bmatrix}
\end{array} = \\
\begin{array}{ccc} b & t & a \\ \begin{bmatrix} 0,8 & 0,5 & 0,2 \\ 0,2 & 0,5 & 0,8 \end{bmatrix}
\end{array}
\end{array}$$

La composición de las relaciones correspondientes a las sentencias a) y b) será:

$$\begin{array}{c}
\begin{array}{cc} T_1 & T_2 \end{array} \\
\begin{array}{c} p \\ v \\ o \\ i \end{array} \begin{bmatrix} 0,7 & 0,3 \\ 0,8 & 0,2 \\ 0,6 & 0,4 \\ 0,2 & 0,8 \end{bmatrix} \cdot \begin{array}{ccc} b & t & a \\ \begin{array}{c} T_1 \\ T_2 \end{array} \begin{bmatrix} 0,8 & 0,5 & 0,2 \\ 0,2 & 0,5 & 0,8 \end{bmatrix}
\end{array} = \begin{array}{ccc} b & t & a \\ \begin{array}{c} p \\ v \\ o \\ i \end{array} \begin{bmatrix} 0,7 & 0,5 & 0,3 \\ 0,8 & 0,5 & 0,2 \\ 0,6 & 0,5 & 0,4 \\ 0,2 & 0,5 & 0,8 \end{bmatrix}
\end{array}$$

4.4.4. Reglas de inferencia borrosas

Consideremos estas dos premisas:

P1: Estamos en una estación no muy fría.

P2: Si la estación es fría, se siente frío, si no, no.

¿Qué conclusión podríamos obtener? Obviamente, una conclusión borrosa referente al grado de frío que se siente en esta estación. De acuerdo con lo visto anteriormente, el significado de P1 es un subconjunto borroso de $U = \{p, v, o, i\}$, y el de P2 es una relación borrosa entre U y $V = \{T_1, T_2\}$. Y la conclusión será un subconjunto borroso de V .

La regla de inferencia borrosa para este caso (que es una generalización del modus ponens) es:

Si R es una relación borrosa entre U y V , y X es un subconjunto borroso de U , el subconjunto borroso inducido en V por X viene dado por la composición

$$Y = X \circ R$$

(tomando X como una relación unaria).

En el ejemplo dado,

$$X = \overline{S^2(\text{fría})} = \{p|0,91; v|0,99; o|0,84; i|0,19\}$$

$$Y = \begin{matrix} & p & v & o & i \\ [0,91 & 0,99 & 0,84 & 0,19] \end{matrix} \begin{matrix} T_1 & T_2 \\ \begin{matrix} p \\ v \\ o \\ i \end{matrix} \begin{bmatrix} 0,7 & 0,3 \\ 0,8 & 0,2 \\ 0,6 & 0,4 \\ 0,2 & 0,8 \end{bmatrix} \end{matrix} = \begin{matrix} T_1 & T_2 \\ [0,8 & 0,4] \end{matrix}$$

Si a las sentencias anteriores se añade: "si se siente frío se usa ropa de abrigo, si no, no", el resultado (grado de uso de las distintas prendas en las estaciones no muy frías) puede calcularse, de acuerdo con los resultados anteriores, así:

$$\begin{matrix} & b & t & a \\ T_1 & T_2 & \begin{bmatrix} 0,8 & 0,5 & 0,2 \\ 0,2 & 0,5 & 0,8 \end{bmatrix} \end{matrix} = \begin{matrix} b & t & a \\ [0,8 & 0,5 & 0,4] \end{matrix}$$

o así:

$$b \quad t \quad a$$

$$\begin{bmatrix} p & v & o & i \\ [0,91 & 0,99 & 0,84 & 0,19] \end{bmatrix} \begin{bmatrix} 0,7 & 0,5 & 0,3 \\ 0,8 & 0,5 & 0,2 \\ 0,6 & 0,5 & 0,4 \\ 0,2 & 0,5 & 0,8 \end{bmatrix} = \begin{bmatrix} b & t & a \\ [0,8 & 0,5 & 0,4] \end{bmatrix}$$

4.4.5. Aproximaciones lingüísticas

Ahora bien, una vez interpretada una sentencia u obtenida una inferencia como hemos visto, lo que tenemos es un subconjunto borroso de un cierto universo. Pero lo que deberíamos obtener sería una sentencia como "se siente bastante frío", "se tiende a usar ropas de no mucho abrigo", etc. Para ello, es preciso asignar a cada subconjunto borroso una aproximación lingüística, que es otro subconjunto borroso que corresponde al significado de una sentencia construida con las partículas "muy", "bastante", etc.

En el último ejemplo habíamos llegado a las conclusiones

$$C1: \{T_1|0,8; T_2|0,4\}$$

$$C2: \{b|0,8; t|0,5; a|0,4\}$$

referentes al frío que se siente y al grado de uso de las ropas de abrigo. Por otra parte, los conjuntos borrosos correspondientes a los significados de "se siente frío" y "la ropa es de abrigo" se habían definido así:

$$\text{"frío"} = \{T_1|0,2; T_2|0,8\}$$

$$\text{"abrigo"} = \{b|0,1; t|0,5; a|0,9\}$$

El significado de "no hace frío" sería el de "frío", que, aproximadamente, se corresponde con el de C1. Para un mejor ajuste, tendríamos que combinar otras partículas (bastante, más bien, etc). Igual ocurre con C2, que, en una primera aproximación, podría asimilarse al complementario de "se usan ropas de abrigo".

5. Resúmen

Hemos pasado revista en este capítulo a algunas de las llamadas "lógicas no clásicas" (hay otras que no hemos mencionado: intuicionista, dinámica, situacional, intensional, epistémica, etc.), entendiendo por tales las que modelan

aspectos de los procesos de razonamiento no contemplados en las lógicas de proposiciones y de predicados (aunque también hemos incluido aquí las lógicas de clases y de relaciones, que no son más que otra manera de contemplar la lógica de predicados).

Entre las que representan ampliaciones de la lógica clásica, es decir, las que conservan todo su lenguaje y su sistema axiomático, hemos comentado la lógica de predicados de orden superior, la lógica modal, la lógica no monótona y la lógica temporal. Y entre las que invalidan ese sistema axiomático, la multivalorada y la borrosa. Esta última, además de contemplar infinitos valores entre la verdad y la falsedad, considera que esos mismos predicados semánticos ("verdadero" y "falso", así como los intermedios) son, en sí mismos, imprecisos. Para modelar esta situación hay una herramienta matemática, la teoría de conjuntos borrosos, cuya base hemos estudiado.

6. Notas histórica y bibliográfica

Al mencionar la lógica de relaciones ha surgido, naturalmente, la referencia a las bases de datos relacionales. El modelo relacional para las bases de datos lo introdujo Codd (1970), y actualmente reviste gran importancia en el campo de la informática aplicada. Para el estudio de este tema hay muchos libros. Recomendamos particularmente los de Ullman (1988, 1989), que cubren no sólo las bases de datos relacionales, sino también las deductivas (que incorporan reglas y mecanismos de inferencia similares a los que estudiaremos en el siguiente capítulo).

A pesar del adjetivo "no clásicas", muchos conceptos (nociones de "modalidades" y de "contingente" como algo que no puede considerarse verdadero ni falso) se remontan a la lógica aristotélica. Pero la formulación moderna de la lógica modal se debe a Lewis (1932), que introdujo los operadores modales de posibilidad y necesidad y la "implicación estricta", y la de la lógica multivalorada a Lukasiewicz (1920, 1930). La formulación de Lukasiewicz sobre lógica multivalorada no es la única. Otros autores prefieren definir de otro modo el significado de las conectivas, como Bochvar (1939) y Kleene (1952).

Las "teorías borrosas" van indisolublemente unidas al nombre de Lofti Zadeh. Fue él quien introdujo los conjuntos borrosos ("fuzzy sets") (Zadeh, 1965), los algoritmos borrosos (Zadeh, 1968), sus aplicaciones biológicas (Zadeh, 1969), sus aplicaciones a sistemas complejos y procesos de decisión (Zadeh, 1973), la lógica borrosa (Zadeh, 1974), la "teoría de la posibilidad" (Zadeh, 1978) (con la que se formaliza la idea de inferencia borrosa de una manera más rigurosa que la que hemos sugerido aquí), las aplicaciones al razonamiento aproximado en general (Zadeh, 1979), y, más recientemente, la metodología de los grafos de reglas borrosas para sistemas expertos (Zadeh, 1992).

Como referencias generales sobre las lógicas no clásicas podemos citar los libros de Haak (1974, 1978), que aborda el tema desde el punto de vista teórico y

filosófico, y el de Turner (1984), que lo hace centrándose en las aplicaciones a la informática.

La lógica modal se ha aplicado tanto en teoría de la programación (Manna y Pnueli, 1979) como en métodos de inferencia no monotónica (McDermott y Doyle, 1980) y en representación del conocimiento (Moore, 1984).

Igualmente, la lógica temporal se ha aplicado para la programación concurrente (Manna y Pnueli, 1981; Manna y Wolper, 1984) y para sistemas de inteligencia artificial (Allen, 1981; McDermott, 1982).

Un texto clásico sobre lógicas multivaloradas es el de Rescher (1969).

Sobre conjuntos borrosos y lógica borrosa hay bastante bibliografía. Recomendamos particularmente tres libros: el de Klir y Folger (1988) y los de Dubois y Prade (1980, 1988). Pero donde más aplicaciones está encontrando la teoría de conjuntos borrosos es en los sistemas de control, y particularmente en el diseño de muchos productos japoneses⁴; tres libros de referencia sobre este campo particular son los de Driankov *et al.* (1993), Yager y Filev (1994) y Kandelk y Langhotz (1994). El número de marzo de 1995 del "Proceedings of the IEEE" (Chand y Chiu, 1995) contiene varios artículos sobre aplicaciones de la lógica borrosa en ingeniería. Además, hay revistas especializadas: "Fuzzy sets and systems", editada por North Holland desde 1978, y "Transactions on fuzzy systems", editada por el I.E.E.E. desde 1993. Aún persiste, sin embargo, bastante controversia sobre la lógica borrosa; a este respecto resulta muy interesante observar la disparidad de opiniones entre los mismos especialistas (Shastri, 1993).

7. Ejercicios

- 7.1. Analizar en lógica de clases los razonamientos expuestos en el ejercicio 7.3 del capítulo 4.
- 7.2. Formalizar los siguientes enunciados, teniendo en cuenta que algunos pueden ser ambiguos (y tener más de una formalización):
 - a) Si es necesario que si p entonces q entonces si es necesario que p entonces es necesario que q .
 - b) No es posible ser valiente u osado si y sólo si no es posible ser valiente y no es posible ser osado.
 - c) Si Pepe siempre se ha mareado cuando ha bebido entonces si mañana bebe se mareará.

⁴ Quizás el lector haya visto algún anuncio de cámaras fotográficas o de vídeo en el que se dice que utilizan "lógica fuzzy" o "lógica gradual". Obviamente, el adjetivo "borroso" no parece muy adecuado para la promoción de este tipo de producto...

- d) Es posible que un programa haya funcionado siempre bien y que mañana no funcione.
- 7.3. Definir la "equivalencia estricta", de modo similar a como se ha definido la implicación estricta en el apartado 2.4.
- 7.4. Analizar mediante tablas de verdad si las leyes de modus ponens y modus tollens son tautologías en lógica trivalorada.
- 7.5. Definir los conjuntos borrosos que sean necesarios en cada caso y analizar los siguientes razonamientos:
- a) *P1*: La mayoría de los hombres son heterosexuales
P2: Sócrates era hombre
C: Es muy posible que Sócrates fuera heterosexual
- b) *P1*: La temperatura es un poco alta
P2: Cuando la temperatura es alta hay que cerrar un poco la válvula
C: Hay que cerrar ligeramente la válvula

Fundamentos de informática

6



6

Aplicaciones en ingeniería del conocimiento

1. Inteligencia artificial e ingeniería del conocimiento

1.1. ¿Qué es la inteligencia artificial?

En este capítulo estudiaremos la aplicación de la lógica a la construcción de "sistemas inteligentes". Empezaremos con algunas consideraciones generales sobre la "inteligencia artificial" (abreviadamente, "IA").

En la historia del pensamiento pueden encontrarse numerosos precursores que, mucho antes de la aparición de las máquinas para el tratamiento de la información,

entrevieron la posibilidad de mecanizar los procesos del razonamiento (por ejemplo, Ramon Llull) o de simular los procesos cognoscitivos (por ejemplo, Torres Quevedo). Ya antes de la aparición del primer ordenador comercial (el "UNIVAC I", que se entregó en 1951), Alan Turing propuso una "prueba" (imaginaria, al estilo de los "experimentos mentales" de los físicos teóricos¹) para determinar la pretendida inteligencia de una máquina. Adaptada y resumida, la "prueba de Turing" podría expresarse así: "póngase a la máquina y a un ser humano en comunicación con un observador (otro ser humano inteligente) a través de sendos terminales con pantalla y teclado mediante los cuales el observador plantea problemas para cuya solución se requiere un cierto grado de inteligencia; si el observador, a la vista exclusivamente de las respuestas, es incapaz de distinguir al ser humano de la máquina, habrá de admitir que ésta posee el mismo grado de inteligencia que el primero, en lo que respecta a la inteligencia necesaria para resolver ese tipo de problemas".

Se da como "fecha de nacimiento" de la inteligencia artificial el 15 de diciembre de 1955, cuando el "Logic Theorist", un programa diseñado por Newell, Shaw y Simon, demostró el teorema 2.15 de los "*Principia Mathematica*" de Whitehead y Russell. En el verano de 1956 se celebró en Darmouth la primera conferencia de IA, en la que se hicieron previsiones muy optimistas (práctica sustitución total de todo el trabajo del hombre por la máquina en un plazo de veinticinco años) que poco después se vieron defraudadas. Los primeros resultados de la IA en productos comerciales no aparecieron hasta la década de los 80.

Pero ¿qué es la IA? No hay una definición universalmente admitida.² Desde los años 50, aparecen ya claramente diferenciados dos objetivos en los trabajos sobre IA: unos están más orientados a las aplicaciones (se define la IA como "la manera de hacer que las máquinas ejecuten tareas inteligentes") y otros hacia la investigación psicológica (se define la IA como "una herramienta para investigar sobre la naturaleza de la inteligencia", o, incluso, como "la ciencia de la inteligencia"). Para lo que aquí nos ocupa, nos quedaremos con la primera definición, pero hay que observar que conduce fácilmente a una concepción evanescente de la IA: antes de la aparición de las calculadoras mecánicas podía pensarse que tales máquinas serían "inteligentes"; hoy nadie diría tal cosa. Del mismo modo, cuando los ordenadores se programaban en lenguaje de máquina, los que estaban definiendo los primeros lenguajes simbólicos y diseñando sus traductores trabajaban en IA (la tarea de traducción tenía que hacerla una persona, y requería "inteligencia"). Y los sistemas expertos, de los que hablaremos más adelante, fueron las primeras aplicaciones prácticas de la IA (olvidado ya el origen, lejano, de los

¹ Ello no obsta para que, desde hace unos años, se organice periódicamente un "concurso" entre programas en el que se premia al que mejor supera la prueba de Turing "engañando" a un equipo de observadores.

² Hay quienes opinan que la inteligencia es un atributo esencialmente humano. Para ellos, la expresión "inteligencia artificial" es, obviamente, una contradicción en sus términos.

primeros traductores de lenguajes), pero hoy pueden considerarse como productos informáticos "normales".

Como se ve, la cuestión conduce, en definitiva, a preguntarse: "¿qué es la inteligencia?", y, por tanto, al segundo tipo de objetivo que hemos mencionado.

A efectos prácticos, y para centrar el tema, comentaremos brevemente dos ideas clave, o *paradigmas*, que impregnan el campo de la IA aplicada: la búsqueda heurística y los sistemas basados en conocimiento (sin entrar en áreas concretas de la IA, como el reconocimiento y síntesis del lenguaje natural y del habla, la visión artificial, la robótica, etc.).

1.2. La búsqueda heurística

En los años 60 se desarrolló en numerosos estudios teóricos y prácticos la idea de utilizar heurísticos para poder abordar problemas que, debido a su *complejidad no polinómica* (concepto que se define en el apartado 3 del capítulo 7 del tema "Algoritmos"), son insolubles de manera algorítmica. (Un ejemplo clásico es el de ciertos juegos, como el del ajedrez, donde la búsqueda de todas las posibilidades desde el inicio de una partida hasta su final requeriría la creación de un árbol con unos 10 elevado a 120 nodos, y un cálculo elemental nos da que una máquina que generase tres nodos por nanosegundo tardaría el descabellado número de 10 elevado a 21 años para generar todo el árbol, y, por tanto, para poder hacer el primer movimiento). El principio de la búsqueda heurística consiste en renunciar a encontrar "la solución", y contentarse con aplicar reglas empíricas o "trucos", llamados *heurísticos*, que la experiencia avala. Los sistemas diseñados al amparo de ese principio permiten así que las máquinas puedan resolver problemas para los que, en principio, no están "preparadas"; el inconveniente es que la máquina deja de ser "infalible": nunca podremos estar seguros de que "su solución" sea "la solución", ni siquiera de que sea "una solución" (en cierto modo, esto la hace también más "humana").

No podemos aquí entrar en los detalles técnicos de cómo se plasma esta idea en algoritmos y programas concretos para ordenador. Al lector interesado le remitimos a los muchos libros publicados sobre IA, algunos de los cuales citamos en el apartado 6.

1.3. Los sistemas expertos y los sistemas basados en conocimiento

Durante los años 70 continuaron los trabajos, especialmente en varias universidades americanas, cada vez más orientados a la resolución de problemas muy específicos, más que al "problema general de la resolución de problemas". Aparecen así el "paradigma del sistema experto" (construir sistemas que emulen a los expertos humanos en actividades muy concretas y muy restringidas) y el "paradigma del conocimiento" (lo que caracteriza a un experto humano es su

conocimiento sobre el problema, más que sus capacidades generales de resolución de problemas).

Se llama *sistema experto* a un sistema informático diseñado para resolver problemas en algún área de aplicación, con una competencia al menos similar a la que pueda tener un experto humano en ese área. Por ejemplo, sistemas ya clásicos son MYCIN, sistema experto para diagnóstico y tratamiento de un número muy reducido de enfermedades infecciosas, PROSPECTOR, sistema experto para determinar la probabilidad de la existencia de yacimientos de ciertos minerales a partir de las evidencias de campo, XCON, para configurar sistemas informáticos con ordenadores VAX y PDP, etc.

Según el caso, el objetivo del sistema experto puede ser el de sustituir al experto humano (lo que puede tener un especial interés en aquellas aplicaciones en las que la experiencia tiene que estar disponible en lugares peligrosos o geográficamente remotos o inaccesibles) o el de ayudar a los expertos humanos a tratar con volúmenes de información que desbordan su capacidad (por ejemplo, en medicina). En cualquier caso, el objetivo final es el mismo de todas las aplicaciones informáticas: relevar a las personas de tareas mecanizables y proporcionarles instrumentos amplificadores de sus capacidades mentales.

Considerado como una "caja negra", un sistema experto deberá satisfacer un criterio similar a la mencionada "prueba de Turing". Es decir, para que un sistema artificial pueda considerarse experto en un área determinada tendrá que resolver problemas al menos con la misma eficacia que lo hace un experto humano en ese área. Así planteado, este criterio conduciría a considerar como expertos a muchos sistemas "convencionales". Pero generalmente se entiende que un sistema experto, además de resolver problemas, tiene que asumir otras funciones características de los expertos humanos. Por ejemplo, poder incrementar sus conocimientos, ya sea por contacto con otros expertos o por experiencia (aspecto éste relacionado con el aprendizaje, que comentaremos más adelante), y poder explicar el razonamiento seguido para resolver un problema.

Acabamos de definir un sistema experto *funcionalmente*, es decir, por "lo que debe hacer". Hablemos ahora de su *estructura*, que guarda una relación estrecha con el "paradigma del conocimiento". En primer lugar, hay que decir que un sistema experto no es más que un programa (o un conjunto de programas) para ordenador. El usuario interactúa con ese programa, que le resuelve, o le ayuda a resolver, sus problemas.

Ahora bien, habitualmente, cuando alguien escribe un programa para resolver una clase de problemas en un ordenador, mezcla en las sentencias de ese programa, de manera más o menos desordenada, los *conocimientos* necesarios para ello y los *procedimientos* que, actuando sobre esos conocimientos, permiten encontrar una solución para cada problema específico. Nada impide, en principio, mantener esa forma de construcción para desarrollar un sistema con unas especificaciones funcionales que satisfagan las características enumeradas más arriba. Pero la necesidad de representar de manera explícita el conocimiento, de adquirirlo a partir de expertos humanos, de acceder a él no sólo para resolver

problemas, sino también para justificar la solución, etc., aconseja la separación de los dos componentes (conocimientos y procedimientos que los manipulan y los usan). Ello conduce a una estructura modular cuyos dos bloques principales son la *base de conocimientos* y el *motor de inferencias*.

Esta concepción modular del sistema, en la que se separan claramente los conocimientos del motor de inferencias, es la que conduce a la denominación de "sistemas basados en conocimiento". En teoría, la definición funcional de "sistema experto" no exige que también sea un "sistema basado en conocimiento" (puesto que, como toda definición funcional, no entra en su estructura), pero en la práctica, todos los sistemas expertos tienen esta arquitectura.

Otro componente de los sistemas basados en conocimientos es la *base de hechos*, que contiene, para cada problema, los datos relativos al mismo (por ejemplo, las manifestaciones de cada paciente y las conclusiones intermedias). Y también son importantes las *interfaces*: la interfaz del experto (para construir, examinar y modificar la base de conocimientos) y la interfaz del usuario final.

1.4. La ingeniería del conocimiento

Es fácil intuir que el primer problema en el diseño de un sistema experto es el de decidir los esquemas para la *representación del conocimiento*. Podemos distinguir varios tipos de conocimiento. Hay un *conocimiento descriptivo*, o *declarativo*, sobre los hechos y relaciones del dominio de experiencia o sobre los datos del problema concreto a resolver (este último constituirá el contenido inicial de la base de hechos). Hay también un *conocimiento procedimental*, o *normativo*, de tipo táctico, que permite obtener conclusiones a partir del conocimiento descriptivo. Y hay, finalmente, un *conocimiento estratégico*, o *de control*, que determina la manera en que se aplica el conocimiento procedimental. Por aclarar ideas sobre un ejemplo muy sencillo, recuérdese el ejemplo 1.7.5 del capítulo 4. En ese caso, el conocimiento descriptivo sería el contenido en los hechos "madre de" y "padre de" que se establezcan, así como en las "reglas" (sentencias condicionales) que definen la relación "antepasado de" en función de "padre de" y "madre de". El procedimental serían las reglas de inferencia, que dicen cómo se obtienen conclusiones, y el estratégico estaría definido en el sistema inferencial: en qué orden y a qué premisas se aplican las reglas de inferencia.

Otro problema es el de la *adquisición del conocimiento*, que puede ser, como es lo más frecuente en los sistemas actuales, a partir de un experto humano (lo cual exige un arduo trabajo para llegar a traducir tal conocimiento, generalmente difícil de explicitar y de naturaleza imprecisa y heurística, al esquema de representación elegido), o bien por autoaprendizaje (inducción del conocimiento a partir de ejemplos resueltos; trataremos este asunto en el apartado 8 del capítulo 5 del tema "Autómatas").

Además está, por supuesto, el problema del diseño del motor de inferencias (que, en principio, contiene tanto conocimiento procedimental como estratégico), dependiente del esquema de representación. Y, finalmente, el problema de la

comunicación con el usuario final: para que el sistema sea aceptable debe "convencer", y para ello debe ser capaz de justificar la línea de razonamiento seguida en cada caso, y, si hace alguna pregunta (es decir, pide algún dato para completar la base de hechos), explicar el motivo de la misma. Todo este conjunto de problemas ha dado lugar a un campo de trabajo que se conoce con el nombre de *ingeniería del conocimiento* (o "ingeniería del saber", o "ingeniería cognoscitiva", o "ingeniería epistemológica").

En el estado actual de la ingeniería del conocimiento, la adquisición se entiende fundamentalmente como una actividad de *modelación*: para diseñar un sistema experto hay que empezar por establecer un modelo de la experiencia (es decir, de la actividad del experto, en términos del conocimiento que aplica para resolver los problemas), un modelo del entorno, de la comunicación, etc. Después, hay que escoger la *arquitectura* más adecuada de acuerdo con ese modelo (la que hemos sugerido, formada por los bloques de base de conocimientos, motor de inferencias, base de hechos e interfaces, es la más primitiva; hay otras más elaboradas, como las basadas en pizarras, las jerárquicas y las de agentes). Y para la implementación del sistema existen *herramientas* (entornos de desarrollo especializados, formados por programas que facilitan todas las actividades de construcción del sistema) muy potentes.

En este capítulo veremos cómo la formulación lógica es un posible (aunque primitivo) modelo para la representación del conocimiento. Para simplificar la exposición, usaremos solamente la lógica de proposiciones. Explicaremos asimismo el funcionamiento de los motores de inferencia (en lógica de proposiciones) y cómo se aborda el problema de la representación del conocimiento impreciso. También daremos una idea de otros esquemas más estructurados para la representación del conocimiento.

2. Sistemas basados en reglas

2.1. Sistemas de producción

Un *sistema de producción* es un modelo de computación que incluye tres componentes: una *base de datos*, un conjunto de *reglas de producción* y un *sistema de control*. Estos tres componentes son, respectivamente, la "base de hechos", la "base de conocimientos" y el "motor de inferencias" del sistema de producción. La llamada "base de datos" no es necesariamente una base de datos en el sentido informático habitual: según el sistema, puede ser desde una sencilla matriz de números hasta una verdadera base de datos. Las reglas de producción se aplican sobre la base de datos, cambiando su estado en cada aplicación, y el sistema de control gobierna esas aplicaciones y hace que la computación se detenga cuando el estado de la base de datos cumple con alguna *condición de terminación* predefinida.

Este es un modelo muy general. Aquí sólo veremos algunos detalles del caso particular en el que las reglas de producción son sentencias condicionales expresadas en lógica de proposiciones, y la base de datos es un conjunto de variables proposicionales.

2.2. Base de datos

En la base de datos están los hechos iniciales y los que se van obteniendo como consecuencias en el proceso inferencial. Las variables proposicionales que contiene pueden estar negadas o no. Es decir, la base de datos es, en nuestro caso, un conjunto de literales (capítulo 2, apartado 2.1). Cada literal representa a un hecho concreto. (En el caso de lógica de predicados, en lugar de variables proposicionales tendríamos predicados aplicados sobre valores concretos de las variables, o sea, sobre constantes). No nos ocuparemos aquí de problemas como el de la estructuración de estos datos, acceso a los mismos, etc.

2.3. Reglas de producción

Las *reglas de producción* (no confundir con las reglas de inferencia, que estarán incluidas en el sistema de control) son pares ordenados (A, B) . Dependiendo de la aplicación concreta, los elementos del par reciben los nombres de "antecedentes" y "consecuente", "condiciones" y "acción" o "premisas" y "conclusión". En nuestro caso particular son, como hemos dicho, sentencias condicionales: $A \rightarrow B$. Como puede adivinarse por los nombres dados a A y a B , A será normalmente una conjunción de literales y B un literal. Es decir, supondremos que nuestras reglas de producción son sentencias de la forma:

$$l_{A1} \wedge l_{A2} \wedge \dots \wedge l_{An} \rightarrow l_B$$

De todos modos, A podría ser una sentencia cualquiera: con unas transformaciones similares a las que hacíamos en el capítulo 2 (apartado 5.5) para llegar a la forma clausulada, puede verse que A siempre se puede transformar en un conjunto de sentencias de este tipo particular. Por su parte, B también podría ser cualquier sentencia. Sólo supondremos, de momento, y por las razones que veremos más adelante (en el apartado 3.1), que no contiene la conectiva " \vee ". Si, por ejemplo, B fuera una conjunción de literales, $l_{B1} \wedge l_{B2} \wedge \dots \wedge l_{Bm}$, la sentencia podría descomponerse en m sentencias con el mismo antecedente y cada una de ellas con uno de los literales como consecuente.

Obsérvese que esta forma de sentencia es la que en el capítulo 2 (apartado 5.6) llamábamos "cláusula de Horn con cabeza", salvo que ahora las variables proposicionales pueden ser literales. (Por otra parte, todo el conjunto de la base de datos puede interpretarse como una cláusula de Horn sin cabeza y negada). Esta forma, aparte de que nos va a simplificar la estructura del sistema de control (o

motor de inferencias), es, además, la que de manera natural se obtiene cuando se le pide a un experto que ponga su conocimiento en forma de reglas.

Por ejemplo, en la base de conocimientos del sistema XCON hay unas 2500 reglas. Una de ellas (traducida a lenguaje natural) es:

si se trata de asignar una fuente de alimentación,
y se ha colocado un módulo SBI en un armario,
y se conoce la posición que ocupa el módulo,
y hay a su lado espacio disponible,
y se dispone de una fuente de alimentación
entonces colocar la fuente en el espacio disponible.

Es claro que, independientemente de lo que signifiquen tales frases en el contexto de conocimiento de XCON, la regla puede formalizarse mediante la sentencia:

$$(p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5) \rightarrow q$$

Otro ejemplo, menos real pero quizás más sugerente, podría ser el de un experto médico que contuviera las reglas:

R1: Si el paciente tiene fiebre
y tose
y tiene dolores musculares
entonces padece gripe

R2: Si el paciente padece gripe o resfriado
y no tiene úlcera gastroduodenal
entonces recomendar aspirina y coñac

La formalización de tales reglas sería:

$$R1: f \wedge t \wedge m \rightarrow g$$

$$R2: (g \vee r) \wedge \neg u \rightarrow a \wedge c$$

Y esta segunda regla puede descomponerse en

$$R2a: g \wedge \neg u \rightarrow a$$

$$R2b: g \wedge \neg u \rightarrow c$$

$$R2c: r \wedge \neg u \rightarrow a$$

$$R2d: r \wedge \neg u \rightarrow c$$

2.4. Sistema de control (o motor de inferencias)

2.4.1. Estrategias hacia adelante y hacia atrás

Suponemos existente una base de conocimientos codificada como un conjunto de reglas de producción, R_1, R_2, \dots del tipo que hemos visto, en las cuales interviene un conjunto de hechos diversos representados por literales, $l_1, l_2, \dots, l'_1, l'_2, \dots$. Ante una situación, se tiene la evidencia de que un subconjunto de esos hechos, l_1, l_2, \dots son verdaderos, y se trata de encontrar qué otros hechos, l'_1, l'_2, \dots pueden inferirse de esa certidumbre y de las reglas, o dicho de otro modo, de encontrar los l'_k tales que

$$l_1 \wedge R_j \rightarrow l'_k$$

sean tautologías.

Utilizando el último ejemplo, si tenemos los hechos f, t, m y $\neg u$, podemos hacer las siguientes inferencias:

$$\frac{f \wedge t \wedge m \quad R1: f \wedge t \wedge m \rightarrow g}{g} \quad (\text{modus ponens})$$

$$\frac{g \wedge \neg u \quad R2a: g \wedge \neg u \rightarrow a}{a} \quad (\text{modus ponens})$$

$$\frac{g \wedge \neg u \quad R2b: g \wedge \neg u \rightarrow c}{c} \quad (\text{modus ponens})$$

(Conclusión: la terapia es aspirina y coñac).

(Planteado en el lenguaje del cálculo de proposiciones, se trataría de demostrar que la sentencia

$$[(f \wedge t \wedge m \wedge \neg u) \wedge (f \wedge t \wedge m \rightarrow g) \wedge (g \wedge \neg u \rightarrow a \wedge c)] \rightarrow (a \wedge c)$$

es un teorema, pero este planteamiento tiene poco interés práctico).

Normalmente, no tendremos dos, sino muchas reglas (en los sistemas expertos es frecuente que sean del orden de cientos o de miles), y, ante unos hechos, hay

dos formas de enfocar el procedimiento de inferencia (es decir, dos *estrategias básicas*):

- (a) Ir aplicando cuantas reglas de producción y cuantas reglas de inferencia se puedan para ir sucesivamente ampliando la base de hechos. Es lo que hemos hecho en el ejemplo, y corresponde a lo que se llama *encadenamiento hacia adelante*. Es también lo que sugeríamos en el capítulo 2 al hablar de la resolución con búsqueda exhaustiva.
- (b) Fijarse un hecho como objetivo y tratar de deducirlo, viendo de qué reglas de producción es consecuente; si alguno de los antecedentes de esas reglas no figura en la base de hechos fijarlo como subobjetivo, etc. Este es el principio del *encadenamiento hacia atrás*.

En ambos casos, el sistema puede llegar a un punto en el que para poder deducir algo le falten hechos no contenidos inicialmente en la base de hechos, y entonces preguntaría sobre ellos al usuario. (Por ejemplo, el sistema: "¿tose mucho el paciente?"; el usuario: "sí", o "no". Obsérvese que es poco natural que el usuario responda así; en los sistemas reales se puede responder en una escala que permite expresar la respuesta de forma más matizada, pero para ello hay que introducir mecanismos de representación del conocimiento impreciso, de los que hablaremos más adelante, en el apartado 3. Aquí nos estamos limitando al caso más sencillo, en el que los hechos sólo pueden ser verdaderos o falsos, que es lo único que permite la lógica clásica.)

Dentro de cada una de estas estrategias básicas caben distintas variantes; es decir, de estrategias concretas.

2.4.2. Un ejemplo

Antes de entrar en la explicación de los algoritmos de inferencia, veamos sobre un ejemplo cómo se aplicarían los procedimientos de encadenamiento hacia adelante y hacia atrás. El ejemplo es abstracto, en el sentido de que partiremos de un conjunto de reglas de producción y de hechos escritos en forma simbólica, prescindiendo de su significado en un contexto de conocimiento.

Supongamos la base de conocimientos constituida por las siguientes reglas de producción:

- | | |
|--------------------------------|---|
| $R1: A \rightarrow C$ | $R6: D \wedge G \rightarrow B$ |
| $R2: A \rightarrow H$ | $R7: C \wedge F \rightarrow B$ |
| $R3: C \rightarrow D$ | $R8: A \wedge H \rightarrow D$ |
| $R4: D \rightarrow E$ | $R9: A \wedge C \wedge H \rightarrow B$ |
| $R5: B \wedge F \rightarrow X$ | $R10: A \wedge B \wedge C \wedge H \rightarrow F$ |

(Aunque en este ejemplo utilicemos A, B, C, \dots en lugar de p_1, p_2, \dots , debe entenderse que se trata de variables proposicionales).

Y supongamos que tenemos como hecho inicial el A , es decir, $BH_0 = \{A\}$. (La base de hechos inicial sólo contiene a A), y que nos planteamos como objetivo el ver si se puede inferir el hecho X .

Con encadenamiento hacia adelante, la estrategia más sencilla es la de ir recorriendo las reglas desde la primera hasta llegar a una que pueda aplicarse (de acuerdo con la regla de inferencia de *modus ponens*), ampliar la base de hechos con la consecuencia, empezar de nuevo con la primera regla, y así sucesivamente hasta incluir el objetivo en la base de hechos, o hasta que ya no puedan aplicarse reglas. En el caso del ejemplo, la $R1$ es aplicable, con lo que la base de hechos se incrementa con C , luego se aplicaría la $R2$, etc. Escribiéndolo resumidamente, en forma de tabla, tendríamos:

ΔBH	A	C	H	D	E	B	F	X
R	1	2	3	4	9	10	5	

donde en la línea superior se indican los hechos con los que sucesivamente se va incrementando la base de hechos, y en la inferior las reglas que en cada momento se aplican. Como vemos, en este caso termina por incluirse X , por lo que del hecho inicial, A , se infiere X .

No es necesario volver a empezar siempre con la primera regla cada vez que se llega a aplicar una de ellas. Otra estrategia, por ejemplo, es la de buscar la regla que contenga más premisas incluidas en BH ; con ella, la inferencia de X en el ejemplo seguiría esta otra secuencia:

ΔBH	A	C	H	B	F	X
R	1	2	9	10	5	

donde vemos que, en el momento en que se dispone de A, C y H en la BH , tras la aplicación de $R1$ y $R2$, ya no se aplica $R3$, sino $R9$, que tiene más premisas.

El otro procedimiento, más natural en un caso como el planteado en este ejemplo, en el que no se trata de derivar cuantas conclusiones se puedan, sino de inferir un objetivo, es el del encadenamiento hacia atrás. Sobre el ejemplo, y expresado informalmente, funcionaría así:

1. Se trata de ver si puede deducirse X . ¿En qué reglas figura como consecuente?
2. Vemos que sólo lo hace en la $R5$. Por tanto, nuestro objetivo (X) se descompone en los subobjetivos que figuran como antecedentes (en forma conjuntiva, es decir, tienen que darse *ambos*) de esa regla (B y F).

4. Para $R6$ hay que deducir D y G . Pero G no figura como consecuente en ningún sitio, por lo que podemos abandonar esta vía y mirar $R7$.

[illegible]**Figura 6.1.**

2.4.3. Estrategias de resolución impulsadas por los hechos

En el ejemplo anterior hemos utilizado siempre la regla de inferencia de *modus ponens*. Como vimos en el capítulo 2 (apartado 5.7), todas las reglas de inferencia pueden resumirse en una, la de *resolución*. Allí estudiamos la resolución en general, aplicada sobre dos cláusulas cualesquiera (generatrices). Vemos que ahora una de las cláusulas es un condicional y la otra un hecho (un literal). Por ejemplo, una vez que la base de hechos contiene B y F , $R5$ nos permite inferir, por *modus ponens*, X . Con la resolución necesitaríamos dos pasos (puesto que B y F son dos cláusulas). Concretamente, $R5$ en forma clausulada sería:

$$\neg B \vee \neg F \vee X$$

Resolviendo con B resultaría:

$$\neg F \vee X$$

y resolviendo con F , quedaría X .

Pero estos dos pasos se pueden encadenar. En efecto,

$$p_1 \wedge p_2 \wedge \dots \wedge p_n \wedge (\neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \vee q) \rightarrow q$$

es una tautología (y, por tanto, una ley, y, consecuentemente, la formalización de una regla de inferencia). Ello nos permite disponer de una regla de resolución adaptada a nuestro caso. En efecto, nuestras reglas de producción son, como hemos dicho, de la forma

$$l_{A1} \wedge l_{A2} \wedge \dots \wedge l_{An} \rightarrow l_B$$

o, puesta en forma clausulada,

$$\neg l_{A1} \vee \neg l_{A2} \vee \dots \vee \neg l_{An} \vee l_B$$

Si la base de hechos contiene $l_{A1}, l_{A2}, \dots, l_{An}$, la ley anterior nos permite inferir l_B .

Decimos que ésta es una *resolución impulsada por los hechos*: es una restricción de la regla de resolución, porque sólo una de las generatrices es una cláusula cualquiera (disyunción de literales), pero, al mismo tiempo, es una ampliación porque admite un número indefinido de otras cláusulas siempre que éstas sean simplemente literales (hechos).

2.4.4. Un algoritmo con encadenamiento hacia adelante

El algoritmo que vamos a presentar en términos muy generales, utilizando pseudocódigo, da por supuesto que todas las reglas de producción están en la forma clausulada.

Diremos que un literal l_i está *demostrado* si figura en la base de hechos (BH). Si figura como tal estará *demostrado como cierto*, y si lo que figura es su negación estará *demostrado como falso*.

Diremos que una regla $l_1 \vee l_2 \vee \dots \vee l_n$ puede *dispararse* si todos los l_i menos uno están demostrados como falsos (y, en tal caso, nuestra regla de inferencia nos permite decir que el que queda es cierto). Una regla *se elimina* si o bien se ha disparado (y en ese caso no sirve más)³ o bien alguno de sus *componentes* (literales) se ha demostrado que es cierto (y en ese caso no nos permite inferir nada). Las reglas *activas* son las que no están eliminadas.

Llamaremos *valor de una regla* al número de componentes no demostradas, si está activa; si una regla está eliminada su valor será 0. El *valor de una componente* (dentro de una determinada regla) será 0 ó 1 según que esa componente figure negada o sin negar, respectivamente, en la regla.

El algoritmo utiliza dos procedimientos. Uno de ellos, al que llamaremos "inferir", recorre todas las reglas para ver si alguna puede dispararse, y, en caso afirmativo, infiere la conclusión, la introduce en la base de hechos, elimina la regla y llama al segundo procedimiento, "actualizar". Lo que hace éste es, dado un hecho, actualizar el valor de todas las reglas en las que figura. Tenemos así:

```

procedimiento inferir;
mientras haya reglas activas
  para cada regla activa
    si valor[regla] = 1 entonces
      conclusión := componente no demostrado;
      valor[conclusión] := valor del componente en la regla;
      introducir conclusión con su valor en la BH;
      valor[regla] := 0;
      actualizar(conclusión);

procedimiento actualizar(hecho);
para cada regla activa
  si regla contiene hecho entonces
    si valor[hecho] = valor del hecho en la regla
      entonces valor[regla] := 0
    si no, valor[regla] := valor[regla]-1;

```

El programa se limitaría a ir leyendo las premisas (base de hechos inicial), y, para cada una, llamar a "actualizar" y luego a "inferir" (que, a su vez, y en su caso, también llamará a "actualizar"):

```

programa encadenamiento_adelante;
mientras haya premisas
  leer(premisa);
  actualizar(premisa);
  inferir

```

³ Obsérvese que ésto sería incorrecto en el caso de utilizar la lógica de predicados.

2.4.5. Un algoritmo con encadenamiento hacia atrás

Este otro algoritmo utiliza dos procedimientos. El llamado "nodoo" resuelve un nodo "O" del árbol, examinando todas las reglas "aplicables" (una regla es aplicable ante un objetivo si contiene un literal que es igual al complemento de ese objetivo); si no puede demostrarlo (variable booleana "demostrado" con el valor falso tras examinar todas las posibilidades), pregunta por él al usuario. El procedimiento "nodoy" se aplica a un conjunto de objetivos (los que corresponden a una ramificación de tipo "Y"), a cada uno de los cuales aplica el "nodoo":

```
procedimiento nodoo(objetivo,demostrado);
demostrado := falso;
si objetivo incluido en BH entonces demostrado := cierto
si no, para todas las reglas aplicables
    mientras demostrado sea falso
        elegir una regla, Ri;
        nodoy(literales de Ri salvo objetivo,demostrado);
si demostrado = falso entonces
    preguntar(objetivo);
    si hay respuesta entonces
        añadirla a la BH;
        demostrado := verdadero;

procedimiento nodoy(conjunto de objetivos,demostrado);
para todos los objetivos del conjunto
    nodoo(objetivo,demostrado);
```

El programa se limitaría a llamar a nodoo con el objetivo final a demostrar y desde ese momento actuarían ya recursivamente los dos procedimientos.

2.5. Notas sobre el uso de la lógica de predicados. La "lógica 0+"

Las limitaciones expresivas de la lógica de proposiciones, comentadas al principio del capítulo 4, se ponen de manifiesto inmediatamente al tratar de utilizarla como modelo para la representación del conocimiento. Para ilustrarlo con un ejemplo muy sencillo, supongamos que la regla "R1" del ejemplo del apartado 2.3 se convierte en esta otra:

Si el paciente tiene fiebre media
y tose bastante
y tiene dolores musculares no localizados
entonces padece gripe

Estamos suponiendo que hay un "objeto" (el paciente) caracterizado por "atributos" (en este caso, síntomas) que pueden tener distintos valores: la fiebre

puede ser media, alta, etc. Si seguimos utilizando lógica de proposiciones tendremos que multiplicar el número de variables proposicionales (fm: fiebre media, fa: fiebre alta, etc.), y, para incluir todo el conocimiento, multiplicar también el número de reglas (de modo que cada una incluya un caso particular). Se comprende fácilmente que en una aplicación real (que contendrá muchos atributos con muchos valores cada uno) esto conduce a un tamaño inadmisibles de la base de conocimientos.

Ya sabemos que la lógica de predicados nos permite expresar conocimientos más generales. Así, si la regla fuese:

Si el paciente tiene fiebre (ligera, media, alta, ...)
y tose (algo, bastante, mucho, ...)
...

la lógica de proposiciones nos exige escribir una regla para cada una de las combinaciones de valores de atributos, pero la de predicados nos permite formalizar ese conocimiento así, por ejemplo:

$$(\forall x) (\forall y) (\forall z) (F(x, y) \wedge T(x, z)) \rightarrow \dots$$

donde $F(x, y)$ se interpreta como "el paciente x tiene fiebre con valor y ", y $T(x, z)$ como "el paciente x tose con intensidad z ".

Por tanto, para las aplicaciones resulta mucho más conveniente tomar la lógica de predicados como modelo para la representación del conocimiento. El inconveniente es que entonces los algoritmos para el motor de inferencias pueden llegar a ser muy ineficientes. Los esquemas que hemos visto anteriormente en pseudocódigo siguen siendo, en esencia, válidos, pero hay que incluir en ellos un mecanismo de unificación (apartado 4.6 del capítulo 4). Como una misma regla es aplicable a individuos u objetos distintos, hay que probarla para cada uno de ellos.

Para mitigar este problema de eficiencia se puede restringir la forma de las reglas y diseñar algoritmos de búsqueda no exhaustiva. Así se hace en Prolog, lenguaje basado en la lógica, que comentaremos en el apartado 2.4.3 del capítulo 5 del tema "Lenguajes". En la mayoría de los sistemas expertos basados en reglas se adopta un enfoque pragmático: se admite que los componentes de las reglas no sean proposiciones, sino que tengan una cierta estructura, aunque no tan general como la que permite la lógica de predicados. Concretamente, utilizan lo que se llaman "*triples objeto-atributo-valor*", y las reglas son de la forma:

Si $\langle O1, A1, V1 \rangle > y \langle O2, A2, V2 \rangle > y \dots$ entonces ...

(si el valor del atributo $A1$ del objeto $O1$ es $V1$ y ...).

Este enfoque, analizado desde un punto de vista formal, equivale a decir que se utiliza una forma restringida de la lógica de predicados. En efecto, la tripla

$\langle O, A, V \rangle$ es equivalente a " $I(A(O), V)$ ", donde I es el predicado "igual" y A es la función "atributo". Todas las reglas se pueden así formalizar en una lógica de predicados que sólo usa ese predicado. Es pues, una forma de la lógica intermedia entre la de predicados (de primer orden) y la de proposiciones (de orden 0), y de ahí que la llamemos "*lógica 0+*".

3. Inferencia plausible

3.1. Fuentes de imprecisión y de incertidumbre

Ocorre con mucha frecuencia que, desde el primer momento en que se empieza a pensar en el diseño de un sistema basado en conocimiento, hay que enfrentarse ineludiblemente al problema de la incertidumbre y de la imprecisión, cuyo origen está tanto en el experto como en el usuario final.

En el experto, cuando tiene que expresar su conocimiento. Si el esquema de representación elegido es el de las reglas de producción (el único que aquí estamos tratando) la incertidumbre y la imprecisión se traducen en la forma de las reglas.

En efecto, decíamos en el apartado 2.3 que las reglas de producción tenían que poder expresarse como sentencias condicionales en cuyo consecuente no figurase la conectiva " \vee ". La justificación es que una sentencia como

$$A \rightarrow q_1 \vee q_2$$

correspondería a una "duda" en el conocimiento de la persona que ha establecido la regla de la que procede esa sentencia: ante la evidencia A se deduce que o bien q_1 o bien q_2 , ¿pero cuál de los dos? Sin embargo, tales dudas son frecuentes cuando se trata de explicitar el conocimiento de los expertos. Así, una de las reglas utilizadas como ejemplo al final del mismo apartado 2.3 sería seguramente más razonable redactarla de este otro modo:

Si el paciente tiene fiebre
y tose
y tiene dolores musculares
entonces
padece gripe
o padece bronquitis
o padece tuberculosis
o ...

Desde luego, reglas de producción expresadas de ese modo no permitirían llegar a ninguna conclusión con una lógica puramente *deductiva*. Hay que introducir, en este caso, un razonamiento *abductivo* (apartado 2.4 del capítulo anterior). Pero hay algo más en la mente del experto (si no lo hubiera, tampoco él

podría concluir nada): ante unas evidencias (signos y síntomas en el caso del diagnóstico médico), puede albergar dudas, pero normalmente "cree" más en unas alternativas que en otras. El problema es cómo cuantificar ese "grado de creencia", y cómo trabajar con él para poder hacer deducciones "plausibles".

La imprecisión e incertidumbre en el usuario aparecen cuando tiene que decir si cierto hecho está presente o no. Por ejemplo, decíamos más arriba (apartado 2.4.1) que parece poco razonable que el usuario tenga que responder "sí" o "no" a una pregunta del sistema sobre si el paciente tose.

"Incertidumbre" e "imprecisión" son dos conceptos diferentes: una proposición es incierta si su valor de verdad o falsedad no se conoce o no se puede determinar, y es imprecisa si se refiere a alguna variable cuyo valor no puede determinarse con exactitud (por tanto, una proposición incierta puede ser precisa, y una imprecisa no ser incierta). En el ejemplo de dado más arriba para la regla, el condicional es incierto, mientras que en el ejemplo de la respuesta del usuario, ésta es imprecisa.

Veremos aquí dos de los mecanismos que se utilizan en los sistemas basados en conocimiento para abordar estos problemas (un tercero lo hemos estudiado en el capítulo anterior: la lógica borrosa). Al nivel introductorio en que nos vamos a mover no será necesario entrar en diferenciaciones entre incertidumbre e imprecisión.

3.2. Inferencia bayesiana

El método bayesiano se siguió en PROSPECTOR y luego lo han adoptado muchos otros sistemas derivados de él. Se basa en la teoría de la probabilidad para cuantificar la imprecisión y/o la incertidumbre y trabajar con ella.

La idea básica consiste en asociar probabilidades a las reglas de producción e identificar tales reglas con probabilidades condicionales. Si llamamos H al suceso consistente en que cierta *hipótesis* sea verdadera y E al consistente en que cierta *evidencia* esté presente, podemos identificar la probabilidad condicional:

$$P(H|E) \text{ (probabilidad de que se dé } H \text{ supuesto } E)$$

con la sentencia condicional:

"si la evidencia E está presente entonces H es verdadera con probabilidad $P(H|E)$ ".

Ahora bien, obsérvese que esto es, precisamente, lo que el experto tiene que inferir: si hay varias hipótesis posibles (por ejemplo, aquí debajo hay un yacimiento de molibdeno, o de cobre, o de oro, o... no hay nada) y se presenta una evidencia E compuesta por otras, $E = E_1$ y E_2 y... (por ejemplo, el terreno es arcilloso y en un arroyo cercano alguien se ha encontrado una pepita de oro y...), de lo que se trata justamente es de llegar a saber cuáles son las probabilidades

$P(H_1|E)$, $P(H_2|E)$,... Si el experto (humano) pudiera darnos todas esas probabilidades para *todas y cada una* de las posibles combinaciones de E , entonces el sistema experto se limitaría a tenerlas almacenadas y a efectuar una búsqueda cuando se le diera una determinada combinación de E . Pero esto es impensable, por el gran número de combinaciones posibles que pueden formar E : no hay experto dispuesto a pasarse años enumerando las distintas posibilidades (terreno arcilloso o no, presencia de ciertas rocas o no, etc.) y dando para cada una de ellas y para cada hipótesis posible una probabilidad.

Lo que sí puede darnos el experto humano son unas estimaciones de las probabilidades *a priori* de cada una de las hipótesis ($P(H_j)$: probabilidad de que se dé H sin saber nada más) y de las probabilidades de que se presenten cada uno de los elementos atómicos de evidencia supuesto que cada una de las hipótesis es verdadera ($P(E_i|H_j)$: probabilidad de que, supuesto que hay molibdeno, el terreno sea arcilloso, etc.). Con esta información, el cálculo de las probabilidades a posteriori, que es lo que interesa, se puede hacer aplicando el conocido teorema de Bayes: Si aparece la evidencia E_i , entonces

$$P(H_j|E_i) = \frac{P(H_j) P(E_i|H_j)}{\sum_j P(H_j) P(E_i|H_j)}$$

Sabemos que una suposición de partida en la demostración de este teorema es que las H_j son mutuamente excluyentes, suposición que, normalmente, es poco razonable en las aplicaciones prácticas (por ejemplo, la existencia de un cierto mineral no excluye la posibilidad de que haya otros). Ahora bien, si conseguimos que el experto humano nos dé no sólo las $P(H_j)$ y las $P(E_i|H_j)$ sino también las $P(E_i|\bar{H}_j)$ podemos escribir el teorema en esta otra forma:

$$P(H_j|E_i) = \frac{P(H_j) P(E_i|H_j)}{P(H_j) P(E_i|H_j) + P(\bar{H}_j) P(E_i|\bar{H}_j)} \quad (1)$$

en la que H_j y \bar{H}_j son siempre mutuamente excluyentes, y $P(\bar{H}_j)$ se puede sustituir por $1 - P(H_j)$.

Análogamente,

$$P(H_j|\bar{E}_i) = \frac{P(H_j) P(\bar{E}_i|H_j)}{P(H_j) P(\bar{E}_i|H_j) + P(\bar{H}_j) P(\bar{E}_i|\bar{H}_j)} \quad (2)$$

donde $P(E_i|\bar{H}_j) = 1 - P(E_i|H_j)$ y $P(\bar{E}_i|\bar{H}_j) = 1 - P(E_i|\bar{H}_j)$

Pero aún no hemos terminado, porque lo normal no es que aparezca un solo átomo aislado de evidencia, E_i , sino varios, de modo que $E = E_1 \cup E_2 \cup \dots$, y lo que queremos calcular es $P(H_j|E)$. En este caso, hay dos modos de proceder:

(a) global: se calculan

$$P(E|H_j) = \prod P(E_i|H_j)$$

tras lo cual se aplica la fórmula anterior; para que ese cálculo fuese correcto sería preciso que todos los E_i fuesen independientes entre sí;

(b) por pasos: para E_1 , se calculan con la fórmula (1) $P(H_j|E_1)$ (si es que E_i es verdadera; si no, se utiliza la fórmula (2)); estos valores se toman como nuevos valores de las probabilidades a priori, $P(H_j)$, y se vuelve a aplicar la fórmula con E_2 , y así sucesivamente. Puede demostrarse que el error cometido en el caso de que no todos los E_i sean independientes es menor que con el procedimiento anterior.

Este método, con algunas variantes que luego explicaremos, constituye el mecanismo inferencial utilizado en PROSPECTOR. Para ayudar a aclarar las ideas, vamos a concretar sobre un ejemplo sencillo: el de la regla a que antes nos referíamos sobre la fiebre, la gripe, etc. Puestos a concretar dando probabilidades, podríamos pensar en descomponer la regla en las nueve siguientes:

- R1: Si tiene fiebre entonces padece gripe con probabilidad 0,5
- R2: Si tiene fiebre entonces padece bronquitis con probabilidad 0,1
- R3: Si tiene fiebre entonces padece tuberculosis con probabilidad 0,4
- R4: Si tose mucho entonces padece gripe con probabilidad 0,1
- R5: Si tose mucho entonces padece bronquitis con probabilidad 0,7
- R6: Si tose mucho entonces padece tuberculosis con probabilidad 0,2
- R7: Si tiene dolores musculares entonces padece gripe con probabilidad 0,7
- R8: Si tiene dolores musculares entonces padece bronquitis con probabilidad 0,2
- R9: Si tiene dolores musculares entonces padece tuberculosis con probabilidad 0,1

Si llamamos E_i a las *evidencias* (en este caso, signos y síntomas: E_1 = tiene fiebre, etc.) y H_j a las *hipótesis* (en este caso, enfermedades: H_1 = gripe, etc.), cada una de las reglas equivale a una probabilidad condicional (por ejemplo, R1: $P(H_1|E_1) = 0,5$: la probabilidad de que tenga gripe supuesto que manifiesta fiebre es 0,5).

Pero obsérvese que no es ésta la información que se precisa para aplicar el mecanismo de inferencia bayesiano. En efecto, ¿qué ocurre si unas evidencias

están presentes y otras no? Es decir, ¿cómo calcular, por ejemplo, $P(H_1|E_1 \text{ y } E_2 \text{ y no } E_3)$? La información que se precisa para aplicar la fórmula de Bayes no es la dada por las $P(H_j|E_i)$, sino por las $P(E_i|H_j)$ (además de las $P(H_j)$).

Por tanto, supongamos que nuestro experto médico (humano) nos ha provisto con los siguientes elementos de conocimiento:

$$P(H_1) = 0,02; P(H_2) = 0,01; P(H_3) = 0,001$$

donde H_1 = gripe, H_2 = bronquitis y H_3 = tuberculosis. (Dicho sea de paso, estas probabilidades a priori son lo que en términos médicos se llaman "prevalencias" de las distintas enfermedades). Y asimismo, llamando E_1 a la fiebre, E_2 a la tos y E_3 a los dolores musculares,

$P(E_1 H_1) = 0,95$	$P(E_1 \bar{H}_1) = 0,01$
$P(E_1 H_2) = 0,8$	$P(E_1 \bar{H}_2) = 0,03$
$P(E_1 H_3) = 0,6$	$P(E_1 \bar{H}_3) = 0,02$
$P(E_2 H_1) = 0,3$	$P(E_2 \bar{H}_1) = 0,03$
$P(E_2 H_2) = 1$	$P(E_2 \bar{H}_2) = 0,01$
$P(E_2 H_3) = 0,8$	$P(E_2 \bar{H}_3) = 0,02$
$P(E_3 H_1) = 0,7$	$P(E_3 \bar{H}_1) = 0,02$
$P(E_3 H_2) = 0,5$	$P(E_3 \bar{H}_2) = 0,01$
$P(E_3 H_3) = 0,4$	$P(E_3 \bar{H}_3) = 0,05$

La interpretación de estas probabilidades condicionales en términos estadísticos es muy fácil: las dos de la primera línea vienen a decir "el 95% de los afectados por la gripe tienen fiebre", y "el 1% de todos los pacientes que *no* tienen gripe presentan fiebre.

¿Cómo se diagnosticaría a un paciente que no presentase fiebre, pero sí tos y también dolores musculares? Procedamos, como hemos indicado, y por pasos. En primer lugar, la ausencia de fiebre nos permite, utilizando la fórmula (2), calcular:

$$P(H_1|\bar{E}_1) = \frac{P(H_1)P(\bar{E}_1|H_1)}{P(H_1)P(\bar{E}_1|H_1) + P(\bar{H}_1)P(\bar{E}_1|\bar{H}_1)} =$$

$$= \frac{0,02 * (1 - 0,95)}{0,02 * (1 - 0,95) + (1 - 0,02) * (1 - 0,01)} = 1,02961 * 10^{-3}$$

y, similarmente, $P(H_2|\bar{E}_1) = 2.0783 * 10^{-3}$; $P(H_3|\bar{E}_1) = 4.084 * 10^{-3}$

Ahora tomamos como nuevos valores de $P(H_j)$ estas probabilidades a posteriori calculadas, y aplicamos la fórmula (1) para tener en cuenta la segunda evidencia (tos), resultando como nuevos valores para $P(H_j)$ los siguientes:

$$P(H_1) = 0,03322; \quad P(H_2) = 0,17237; \quad P(H_3) = 0,03956$$

Y, finalmente, considerando que E_3 también está presente, otra aplicación de la fórmula (1) nos conduce a:

$$P(H_1) = 0,78; \quad P(H_2) = 0,91; \quad P(H_3) = 0,25$$

Es decir, se infiere que la enfermedad más probable es la bronquitis, seguida de cerca por la gripe.

Resumiendo, el conocimiento del experto humano quedará codificado en la base de conocimientos por las probabilidades a priori de cada una de los posibles resultados, o hipótesis, $P(H_j)$, y por las probabilidades condicionales de cada una de las evidencias elementales a cada una de las hipótesis, $P(E_i|H_j)$ y $P(\bar{E}_i|\bar{H}_j)$. El sistema preguntará al usuario por algún elemento de evidencia, E_i , y el mecanismo de inferencia calcula las probabilidades a posteriori, $P(H_j|E_i)$ de acuerdo con la fórmula explicada, valores que sustituyen a los previos de $P(H_j)$; luego el sistema preguntará por otro E_i , y así sucesivamente.

¿Cuándo pregunta el sistema por un elemento de evidencia u otro, y cuándo se detiene el proceso? La idea básica es la siguiente: a la vista de las $P(H_j)$ actualizadas en cada momento, y de todos los valores de $P(E_i|H_j)$ y de $P(\bar{E}_i|\bar{H}_j)$ el sistema calcula cuál de las E_i tiene más influencia (teniendo en cuenta que la respuesta puede ser positiva o negativa) en las posibles modificaciones de las $P(H_j)$, y pregunta por ella. Pero si las posibles modificaciones son tales que ninguna de las $P(H_j)$ resultantes puede llegar a ser mayor que la $P(H_j)$ que actualmente es máxima, entonces el proceso termina con H_j como hipótesis más probable. Si no se hiciera así, el usuario estaría obligado a dar todas las E_i cada vez que hiciese una consulta.

Hemos dicho al principio que éste es el mecanismo de inferencia básico de PROSPECTOR, y así es, pero con una variante que permite que el experto humano pueda comunicar su conocimiento de forma algo más cómoda que dando probabilidades condicionales. Esta variante utiliza la "verosimilitud" ("odds"): si la probabilidad de un suceso es P , su verosimilitud es $V = P/(1-P)$ (por tanto, V es un número comprendido entre cero e infinito). Sustituyendo P por $V/(1+V)$ en las fórmulas de Bayes (1) y (2) y operando, resulta:

$$V(H_j|E_i) = MS_{ij} * V(H_j) \quad (3)$$

$$\text{y } V(H_j|\bar{E}_i) = MN_{ij} * V(H_j), \quad (4)$$

con

$$MS_{ij} = P(E_i|H_j) / P(E_i|\bar{H}_j) \quad (\text{medida de suficiencia})$$

$$MN_{ij} = P(\bar{E}_i|H_j) / P(\bar{E}_i|\bar{H}_j) \quad (\text{medida de necesidad})$$

Puede comprobarse (teniendo en cuenta las leyes de las probabilidades) que tiene que cumplirse que si una es mayor o igual que 1 la otra tiene que ser menor o igual que 1; para homogeneizar, se definen las E_i de tal modo que siempre resulte $MS_{ij} \geq 1$ y $MN_{ij} \leq 1$.

Entonces, MS_{ij} estará comprendida entre uno e infinito, y su interpretación es la siguiente: si $MS_{ij} = 1$, de acuerdo con (3) la verosimilitud de la hipótesis H_j tras saber que se da la evidencia E_i es la misma que antes de saberlo, por lo que en este caso es indiferente conocer E_i o no, mientras que si MS_{ij} tiende a infinito, la verosimilitud a posteriori de H_j tiende también a infinito cualquiera que sea su verosimilitud a priori, es decir, E_i es lógicamente suficiente para inferir H_j ; por tanto, MS_{ij} es, como su nombre indica, una medida de la suficiencia del conocimiento de E_i para la inferencia de H_j , que el experto humano ha de evaluar entre uno e infinito.

En cuanto a MN_{ij} , si vale 1, según (4) la ausencia de E_i no afecta a la verosimilitud de la hipótesis, es decir, no es en absoluto necesario que E_i esté presente, mientras que si vale 0 la ausencia de E_i reduce a cero la verosimilitud de H_j ; E_i sería lógicamente necesaria. El experto humano habrá de evaluar, en una escala de 0 a 1 la necesidad de que E_i esté presente para inferir H_j .

En resumen, la base de conocimientos constaría de las probabilidades a priori de cada hipótesis y de un conjunto de reglas que se pueden esquematizar así:

$$H_j \rightarrow E_i (MS_{ij}, MN_{ij})$$

(es decir, reglas de producción con dos números asociados). Y la inferencia consiste en ir recopilando E_i y actualizando las probabilidades de cada hipótesis.

Hay otro asunto a considerar: el de la incertidumbre en el usuario cuando se le pregunta por la existencia o no de un determinado elemento de evidencia, E_i . Según (3) y (4), si la respuesta es que E_i está presente (con seguridad), entonces $V(H_j|E_i) = MS_{ij} * V(H_j)$, mientras que si no hay duda de que E_i no está presente, entonces $V(H_j|\bar{E}_i) = MN_{ij} * V(H_j)$. Pues bien, lo que se hace es permitir al usuario que responda en una escala entre +5 y -5. Una respuesta +5 equivale a una seguridad absoluta en la presencia de E_i , y se aplicaría la primera fórmula. Una respuesta de -5 (seguridad absoluta de que E_i está ausente) conduciría a aplicar la segunda fórmula. Una respuesta 0 (ignorancia total) dejaría inalterada

$V(H_j|E_i)$. Y para valores de R intermedios, el sistema hace una interpolación entre MS_{ij} , 1 y MN_{ij} . Si, por ejemplo, se hacen interpolaciones lineales, tendríamos las siguientes fórmulas:

$$si \ R > 0, M_{ij} = \frac{R * (MS_{ij} - 1) + 5}{5}$$

$$si \ R = 0, M_{ij} = 1$$

$$si \ R < 0, M_{ij} = \frac{R * (1 - MN_{ij}) + 5}{5}$$

y actualizaríamos así la verosimilitud:

$$V(H_j|E_i) = M_{ij} * V(H_j)$$

Con esto cerramos este apartado, en el que hemos visto el *principio* de la inferencia bayesiana. En realidad, los sistemas expertos que siguen este principio son algo más complejos. Por ejemplo, en PROSPECTOR, según lo que hemos dicho, cada "hipótesis" debería corresponder a la existencia un determinado mineral. Pero, en realidad, para cada mineral hay una jerarquía de hipótesis que se combinan formando una red, y en esta red hay también relaciones lógicas imprecisas, para las que se aplican las "leyes de Lukasiewicz" que veíamos en el capítulo 5 (apartado 3). Este esquema particular de PROSPECTOR se ha generalizado y formalizado en lo que se llaman "redes de inferencia probabilística" o "redes de creencias".

3.3. Inferencia mediante factores de certidumbre

Una objeción que puede hacerse al mecanismo explicado en el apartado anterior es que utiliza una base matemática rigurosa (la teoría de la probabilidad) para aplicarla a unas "creencias subjetivas" que no cumplen las leyes de las probabilidades. Hay otros marcos teóricos (elaborados, como el bayesiano, antes de la aparición de los sistemas expertos) más orientados al tratamiento de la subjetividad. Por ejemplo, la teoría de la evidencia de Dempster-Shafer (de difícil aplicación práctica en los sistemas expertos), o la lógica borrosa (estudiada en el capítulo anterior, y cuyo futuro parece más claro).

Pero también existen enfoques más heurísticos (no basados en teoría formal alguna). El más conocido, y más utilizado hasta ahora, es el que se introdujo en el diseño del sistema MYCIN, que pasamos a resumir.

Las reglas de producción se expresan en la forma:

$$E_i \rightarrow H_j (C_{ij})$$

donde C_{ij} es el *factor de certidumbre* de la regla⁴, número comprendido entre -1 y +1 que expresa el "grado de confianza" en esa regla: supuesto que E_i sea verdadero, $C_{ij} = +1$ correspondería a una seguridad absoluta de que se deduce H_j , y $C_{ij} = -1$ a una seguridad absoluta de que H_j es falsa ($C_{ij} = 0$ correspondería a una incertidumbre o ignorancia total sobre el asunto).

Por ejemplo, una de las aproximadamente quinientas reglas contenidas en la base de conocimientos de MYCIN era la siguiente:

Si la infección es bacteriemia primaria
y la toma del material a cultivar es una toma estéril
y se cree que la puerta de entrada del organismo es el
tracto gastrointestinal,
entonces hay bastante evidencia (0,7) de que la identidad
del organismo sea bacteroides.

El número 0,7 que figura en el consecuente es el factor de certidumbre asociado a esa regla.

Los distintos elementos de evidencia pueden también llevar asociados factores de certidumbre (que introducirá el usuario cuando el sistema le pregunte por esos elementos de evidencia).

El mecanismo inferencial de MYCIN consiste en un algoritmo de encañamiento hacia atrás como el explicado en el apartado 2.4.5, al que se añaden algunos heurísticos⁵ para ir combinando los factores de certidumbre y terminar dando una certidumbre final a cada una de las posibles hipótesis. Los heurísticos principales son:

- Si existe la regla $A \rightarrow B (C_R)$ y se ha calculado un factor de certidumbre C_A para A , si $C_A < 0$, la regla no se aplica, y en caso contrario, se asigna a B un factor $C_B = C_A * C_R$.
- En general, A estará compuesto por otros hechos unidos por conectivas \vee , \wedge , y \neg ; el cálculo del factor de certidumbre resultante se hace de acuerdo con las fórmulas:

$$C(A1 \vee A2) = \text{máx}(C_{A1}, C_{A2})$$

$$C(A1 \wedge A2) = \text{mín}(C_{A1}, C_{A2})$$

$$C(\neg A) = -C(A)$$

⁴ Quizás fuera mejor llamarle "factor de incertidumbre": "certidumbre", sinónimo de "certeza", indica una seguridad absoluta, y no parece adecuado graduarla mediante un "factor".

⁵ Recuérdese (apartado 1.2) que, en las técnicas de inteligencia artificial, se llama heurístico a cualquier "truco", o regla empírica, que se ha comprobado que sirve de ayuda en la resolución de un problema.

- En los nodos "Y", de acuerdo con los heurísticos anteriores, el C de la regla se multiplica por el menor de los C de las premisas.
- En los nodos "O", si sólo hay dos ramas para cuyos elementos se han calculado C_1 y C_2 , el factor de certidumbre del resultado es:

$$C = C_1 + C_2 - C_1 * C_2 \quad \text{si } C_1 * C_2 > 0$$

$$C = \frac{C_1 + C_2}{1 - \min(|C_1|, |C_2|)} \quad \text{si } C_1 * C_2 < 0$$

Si hay más de dos ramas, se calcula el C para las dos primeras, el resultado se combina con el de la tercera, etc.

- Siempre que como consecuencia de un cálculo resulte $|C| \leq 0,2$, se hace $C=0$. (Esto acelera los algoritmos y hace más claras las explicaciones y justificaciones del sistema).

Para ilustrar con un caso simplificado cómo se aplican estos heurísticos, volvamos a nuestro ejemplo de la fiebre, la gripe, etc. Supongamos que la base de conocimientos está formada por las nueve reglas enumeradas en el apartado 3.2, pero interpretando que lo que allí se llaman "probabilidades" son factores de certidumbre. Es decir:

$$R1: f \rightarrow g (0,5) \quad R4: ts \rightarrow g (0,1) \quad R7: d \rightarrow g (0,7)$$

$$R2: f \rightarrow b (0,1) \quad R5: ts \rightarrow b (0,7) \quad R8: d \rightarrow b (0,2)$$

$$R3: f \rightarrow tb (0,4) \quad R6: ts \rightarrow tb (0,2) \quad R9: d \rightarrow tb (0,1)$$

Y supongamos que la evidencia presente es: fiebre(-0,8), tos(0,9), dolores(1). (Bastante seguro que no tiene fiebre, casi seguro que tiene tos y con seguridad que tiene dolores musculares).

Para cada una de las tres hipótesis (gripe, bronquitis, tuberculosis) tenemos un árbol muy sencillo: un simple nodo "O" con tres ramas. Para la primera (gripe) resulta:

$$\text{Rama1 (R1): } -0,8 * 0,5 = -0,4$$

$$\text{Rama2 (R4): } 0,9 * 0,1 = 0,09$$

$$\text{Rama3 (R7): } 1 * 0,7 = 0,7$$

Combinando la Rama1 con la 2,

$$\frac{-0,4 + 0,09}{1 - 0,09} = -0,34$$

y combinando este factor de certidumbre con el de la Rama3,

$$C_g = \frac{0,7 - 0,34}{1 - 0,34} = 0,54$$

Procediendo de igual modo con las otras hipótesis se obtiene:

$$C_b = 0,68, \text{ y } C_{tb} = -0,08,$$

es decir, algo más de certidumbre en que sea bronquitis que gripe, y una incertidumbre prácticamente absoluta en cuanto a que pueda o no ser tuberculosis.

4. Otros esquemas para la representación del conocimiento

El objetivo de este capítulo era ver un campo de aplicación de la lógica: los sistemas basados en conocimiento. Pero en muchos de estos sistemas se utilizan otras técnicas más estructuradas para la representación del conocimiento, y parece oportuno, para completar el capítulo, dar una idea de dos de las más conocidas: las redes semánticas y los marcos ("frames").

Una preocupación común en todas estas técnicas es la de tener una representación lo más estructurada posible a fin de facilitar el almacenamiento, modificación y búsqueda en las bases de conocimiento. Vamos a ver sobre un ejemplo cómo de la representación lógica puede pasarse a otras representaciones más estructuradas.

Supongamos que tenemos los siguientes hechos:

- La Memoria Transfiere Datos e Instrucciones al Procesador
- El Procesador Transfiere Datos a la Memoria
- El Procesador Interpreta Instrucciones
- La Memoria Almacena Datos e Instrucciones

Pensando en lógica de predicados, podemos definir dos predicados ternarios: $T(x,y,z)$ para representar " x transfiere z a y " y $F(x,y,z)$ para " x es hacer y con z ". Nuestros hechos se traducirán entonces a predicados aplicados sobre las constantes M (Memoria), P (Procesador), D (Datos), I (Instrucciones), R (Interpreta) y A (Almacena). (También podríamos utilizar letras minúsculas, si quisiéramos seguir al pie de la letra los convenios del capítulo 4 para los nombres de las constantes).

La representación en lógica de predicados sería:

$$T(M,P,D); T(P,M,D); T(M,P,I); \\ F(P,R,I); F(M,A,I); F(M,A,D).$$

Esta base de hechos es pequeña, pero en un caso real podría contener cientos de predicados, y para mejorar el acceso convendría estructurarla. Por ejemplo, podríamos agrupar separadamente (aun cuando aparezcan repeticiones) los hechos referentes a la memoria y los referentes al procesador:

Memoria:

$T(M,P,D); T(P,M,D); T(M,P,I); F(M,A,I); F(M,A,D)$

Procesador:

$T(M,P,D); T(P,M,D); T(M,P,I); F(P,R,I)$

Ahora los hechos están indexados por objetos del dominio del discurso. Se dice que es una representación *centrada en los objetos*, en este caso, en los objetos físicos. También podemos adoptar una representación centrada en objetos abstractos: transferencias y funciones.

Todos los predicados de este ejemplo son ternarios. Pero en las representaciones estructuradas es preferible trabajar sólo con predicados binarios. La razón es que si se quiere perfeccionar el conocimiento diciendo, por ejemplo, que la transferencia de datos de la memoria al procesador se hace a través del bus de datos, habría que convertir el predicado ternario en otro cuaternario, y habría que modificar también los procedimientos de inferencia. Vamos a ver que es posible expresar todo con predicados binarios y conseguir un sistema más modular y fácil de actualizar.

Definamos, para nuestro ejemplo, dos conjuntos, o *clases*: {transferencias} y {funciones}. Todo lo que se dice sobre transferencias y funciones puede expresarse mediante predicados binarios que relacionan a los argumentos de los predicados ternarios originales con un elemento arbitrario de esas clases. Por ejemplo, la fórmula atómica $T(M,P,D)$ se transformará en la sentencia

$$(\exists x)(\text{Pertenece}(x, \{\text{transferencias}\}) \wedge \text{Fuente}(x, M) \text{Destino}(x, P) \wedge \text{Objeto}(x, D))$$

y la $F(P,R,I)$, en

$$(\exists x)(\text{Pertenece}(x, \{\text{funciones}\}) \wedge \text{Unidad}(x, P) \wedge \text{Función}(x, R) \wedge \text{Objeto}(x, D))$$

Los cuantificadores existenciales pueden eliminarse creando constantes de Skolem (capítulo 4, apartado 4.3): $T1$ en sustitución de x en la primera sentencia, y $F1$ en la segunda. Por otra parte, los predicados binarios que relacionan los argumentos originales con algún elemento arbitrario de las clases también pueden expresarse como funciones definidas sobre esas clases. Es decir, en vez de escribir, por ejemplo, " $\text{Fuente}(x, M)$ ", podemos definir la función "fuente" y escribir " $\text{fuente}(x) = M$ ". Obsérvese el cambio de notación, pero no de contenido semántico: "Fuente" es un predicado; $\text{Fuente}(T1, M)$ se evalúa como verdadero o falso; "fuente" es una función; $\text{fuente}(T1)$ es un individuo; "=" es un predicado

binario ("fuente(x) = M " es lo mismo que "Igual(fuente($T1$), M)"), que se evalúa como verdadero o falso.

Haciendo esas dos transformaciones, las sentencias anteriores se escribirán así:

$\text{Pertenece}(T1, \{\text{transferencias}\}) \wedge (\text{fuente}(T1)=M) \wedge (\text{destino}(T1)=M) \wedge (\text{objeto}(T1)=D)$

$\text{Pertenece}(F1, \{\text{funciones}\}) \wedge (\text{unidad}(F1)=P) \wedge (\text{función}(F1)=R) \wedge \text{objeto}(F1) = D)$

Esta representación puede parecer, de momento, bastante más farragosa que la que inicialmente teníamos, pero obsérvense dos cosas: que hemos limitado a dos los predicados y que éstos son binarios (si bien a costa de introducir arbitrariamente funciones) y que si, como decíamos antes, se quieren ampliar las relaciones con nuevos elementos del dominio del discurso (por ejemplo, buses) basta con definir nuevas funciones sobre las clases de base y seguir utilizando los mismos predicados binarios "Pertenece" e "Igual".

Poniendo en grupos separados todos los hechos que identifican a $T1$, $T2$, etc., tendremos una representación más estructurada:

$T1$
 $\text{Pertenece}(T1, \{\text{transferencias}\})$
 $\text{fuente}(T1) = M$
 $\text{destino}(T1) = P$
 $\text{objeto}(T1) = D$

etc.

Como ahora todas las funciones figuran dentro de un grupo identificado por su argumento, no hace falta especificar éste, y escribiremos "fuente: M ", etc. Además, para abreviar el predicado " $\text{Pertenece}(T1, \{\text{transferencias}\})$ " escribiremos "tipo: transferencias". Llegamos así a la representación estructurada:

$T1$
 tipo: transferencias
 fuente: M
 destino: P
 objeto: D

$T2$
 tipo: transferencias
 fuente: P
 destino: M
 objeto: D

T3
tipo: transferencias
fuente: *M*
destino: *P*
objeto: *I*

F1
tipo: funciones
unidad: *P*
función: *R*
objeto: *I*

F2
tipo: funciones
unidad: *M*
función: *A*
objeto: *I*

F3
tipo: funciones
unidad: *M*
función: *A*
objeto: *D*

Se dice que *T1*, *T2* y *T3* son *casos* o *ejemplares* ("instances") del *marco* ("frame") general *T*, y *F1*, *F2* y *F3* lo son del marco *F*. Cada marco tiene como componentes *ranuras* ("slots") de la forma nombre_ranura: valor_ranura.

Hay informaciones implícitas sobre pertenencia a clases en nuestra representación original que podemos añadir ahora:

M
tipo: unidades

P
tipo: unidades

R
tipo: funciones

A
tipo: funciones

D
tipo: informaciones

I
tipo: informaciones

Una *red semántica* es un grafo en el que los nodos pueden representar objetos, conceptos o conjuntos y los arcos relaciones entre ellos. Para nuestro ejemplo, la red semántica podría ser la dibujada en la figura 6.2.

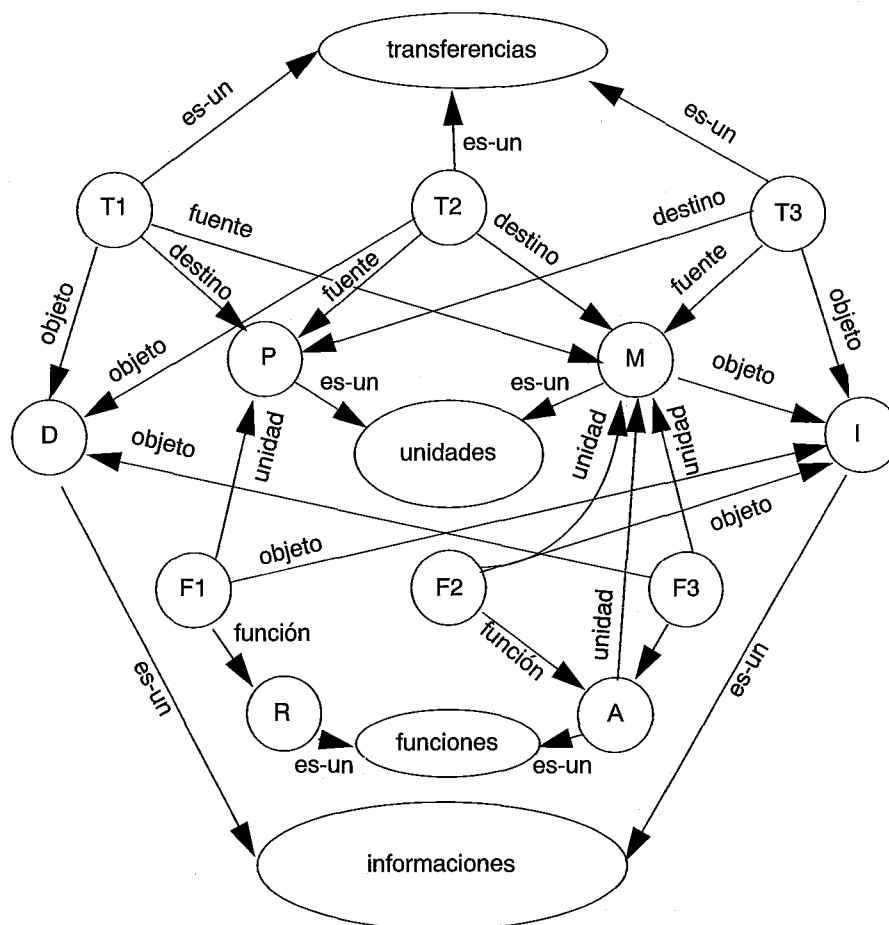


Figura 6.2.

Aquí solamente hemos introducido el principio de las representaciones estructuradas. En el ejemplo, lo que hemos representado es conocimiento declarativo. Los marcos hay que ampliarlos con nuevos conceptos para representar el conocimiento procedimental, y hay que utilizar métodos inferenciales especiales que operen sobre esos marcos. De nuevo remitimos al lector interesado a la bibliografía sobre el tema.

Sólo señalaremos, para terminar, la coincidencia entre los conceptos que animan a los marcos y las redes semánticas y los que subyacen en la programación

orientada a objetos, cuyos fundamentos estudiaremos en el tema "Algoritmos" (capítulo 4, apartado 5).

5. Resumen

La lógica formal es una herramienta adecuada para la representación del conocimiento declarativo, procedimental y de control y, por tanto, para el diseño de sistemas basados en conocimiento y sistemas expertos.

Uno de los modelos más utilizados para el diseño de sistemas basados en conocimiento es el de los *sistemas basados en reglas*. Las reglas pueden formalizarse como sentencias condicionales, y, por tanto, todo lo que la lógica nos enseña sobre sistemas inferenciales es aquí aplicable.

Los expertos humanos suelen trabajar con reglas no muy bien definidas y con elementos de evidencia imprecisos o inciertos. Para diseñar un sistema experto es preciso modelar esos aspectos de la actividad humana, y para ello existen varias técnicas: factores de certidumbre, probabilidades, lógica borrosa, etc.

Hay otras técnicas para representar el conocimiento de una manera más estructurada que con la lógica. Las más conocidas son los *marcos* ("frames") y las *redes semánticas*.

6. Notas histórica y bibliográfica

El artículo de Alan Turing donde se describe la famosa "prueba" puede encontrarse traducido en Pylyshyn (1975), libro que reproduce numerosos trabajos pioneros sobre informática (Hay una edición más moderna y más completa, pero no traducida: Pylyshyn y Bannon, 1989).

El paradigma de los sistemas basados en conocimiento comenzó a tomar cuerpo a finales de los años 60. Suelen citarse como "precursores" los programas DENDRAL y MACSYMA. El primero, desarrollado en la Universidad de Stanford, permitía deducir la estructura química molecular de un compuesto orgánico a partir de su fórmula y de datos espectrográficos y de resonancia magnética nuclear (Buchanan et al., 1969); hay una versión comercial del mismo que utilizan varias empresas farmacéuticas. MACSYMA, del M.I.T., realizaba cálculo diferencial e integral mediante manipulación simbólica de expresiones algebraicas (Martin y Fateman, 1971), y ha derivado también en productos comercializados. Los sistemas expertos "pioneros" más conocidos son MYCIN, para diagnóstico y tratamiento de enfermedades infecciosas (Shortliffe, 1976) e INTERNIST, para diagnóstico en medicina interna (Pople et al., 1975).

En los años 80 se produjo una "industrialización" de los sistemas expertos, en la que empresas ya establecidas se interesan por este nuevo campo, se creaban otras dedicadas exclusivamente a él, y aparecían en el mercado herramientas

software para el desarrollo de sistemas expertos. Actualmente podemos más bien hablar de una "asimilación": la arquitectura de los sistemas basados en conocimiento, y, particularmente, la basada en "agentes inteligentes" está plenamente integrada entre las técnicas de la ingeniería del software (y raramente se la relaciona ya con la "inteligencia artificial").

Post (1943) fue el primero que, desde un punto de vista teórico, formuló el modelo de "sistema de producción" como un mecanismo computacional general. Las redes semánticas se propusieron inicialmente como modelo de la memoria humana (Quillian, 1968), y se han utilizado en los sistemas expertos PROSPECTOR (Duda *et al.*, 1978), CASNET (Weiss *et al.*, 1978), CADUCEUS (Myers *et al.*, 1982), IRIS (Trigoboff y Kulikowski, 1981), etc., y los marcos (frames) fueron introducidas por Minsky (1975) con idea parecida ("cuando a la mente se le plantea una situación nueva, busca en la memoria alguna estructura estereotipada de información"), y en ellas se han basado otros diseños, como los sistemas PIP (Pauker *et al.*, 1976), RADEX (Chandrasekaran *et al.*, 1980), CENTAUR (Aikins, 1983), etc., y otras representaciones estructuradas, como las *escenas* (scripts), de Schank y Abelson (1977).

Hayes (1977, 1979) puso de manifiesto las relaciones entre las representaciones en lógica de predicados y mediante estructuras. En la presentación de estas relaciones en el apartado 4 hemos seguido a Nilsson (1987).

Como textos generales sobre inteligencia artificial, aparte del ya citado de Nilsson (que está traducido), teórico y académico, otro clásico, pero más práctico, es el de Charniak y McDermott (1985), y, más moderno (y también traducido), el de Rich y Knight (1994). Sobre el debate de si la IA es posible, todos los que se sienten inclinados a responder inmediatamente "sí" deberían leer el libro de Kelly (1993), que contiene una reflexión crítica profunda sobre la cuestión.

Sobre sistemas expertos, el libro de Buchanan y Shortliffe (1984) describe con todo detalle MYCIN, EMYCIN y demás desarrollos del "Stanford Heuristic Programming Project". Un texto detallado y orientado a la programación con OPS5 (lenguaje orientado al desarrollo de sistemas de producción) es el de Brownston *et al.* (1985). Más fundamental, sobre sistemas basados en conocimiento en general, es el de Frost (1986). En estos libros se pueden estudiar los principios y modelos básicos de los sistemas expertos. Sobre aspectos más prácticos de la ingeniería del conocimiento (selección de herramientas, gestión de proyectos, etc.) trata el de Prerau (1990). El de Durkin (1993) contiene un catálogo de cientos de aplicaciones de los sistemas expertos. Hay muchas publicaciones periódicas sobre este tema. Una de las más conocidas es el "IEEE Expert".

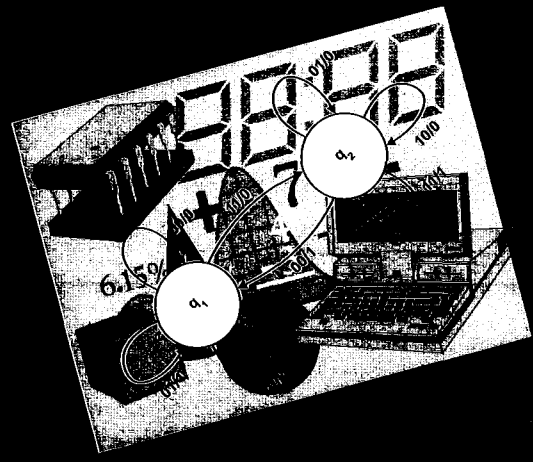
De la incertidumbre en IA trata el libro de Kanal y Lemmer (1986), y del razonamiento aproximado el de López de Mántaras (1990). La teoría de la evidencia se puede estudiar en el libro de Shaffer (1976), pero, como decíamos en el apartado 3.3, no se está aplicando en el diseño de sistemas expertos. Los otros dos marcos teóricos son la teoría de la posibilidad y la teoría de las redes de creencias. Para el primero, remitimos a la bibliografía citada en el apartado 6 del capítulo 5, y para el segundo, al texto de Pearl (1988).

Parte II

Autómatas

Fundamentos de informática

1



Ideas generales

1. Autómatas e información

La palabra "autómata", en el lenguaje ordinario, normalmente evoca algo que pretende imitar funciones propias de los seres vivos, especialmente las relacionadas con el movimiento. (Véase, para corroborar esta aserción, la definición de un diccionario cualquiera.) En este sentido, un ejemplo de autómata sería el típico robot antropomorfo o zoomorfo dotado de capacidades autónomas de movimiento que le permiten ejecutar las órdenes o seguir el programa establecido por un ser inteligente.

En el campo de la Informática lo fundamental no es la simulación del movimiento, sino la simulación de los procesos de tratar la información, y el ejemplo típico de autómata no es ya el robot mecánico sino el ordenador.

Pensemos en la naturaleza del trabajo que realizan los ordenadores. Bastan unos conocimientos básicos de informática para llegar a conclusión de que un ordenador no es más que un dispositivo que manipula símbolos. Un ejemplo será útil para reforzar esta idea:

Consideramos un ordenador con registros de dieciséis bits. Supongamos que este ordenador recibe de un periférico una serie de impulsos que se graban en un registro dejando en él la siguiente configuración de bits:

1010010001000001

¿Qué significa esto para el ordenador? Entre otras muchas cosa puede ser:

- El número -23487 expresado en binario con convenio de complemento a 2.
- Los caracteres "ñ,ü" codificados en código ASCII ampliado a 8 bits.
- Una instrucción del lenguaje de máquina del ordenador.

El que sea una u otra cosa depende de dos personas:

- El diseñador del ordenador (para ser realista habría que hablar del equipo de diseño), que decidió que los números se representen en binario y complemento a 2, o que, interpretado como instrucción, ese código de operación signifique, por ejemplo, "sumar", y no otra cosa, etc.,
- El usuario (en este caso, el programador), que decide que en momento dado el ordenador lleve esa configuración de bits de la memoria al acumulador, o al registro de instrucción, o a la unidad de salida, etc. Al tomar tal decisión, el usuario está dotando al conjunto de bits de una significación y, por consiguiente, de una capacidad para representar información. Para el ordenador, sin embargo, los bits no son más que símbolos materializados por los niveles de tensión en los circuitos.

A veces se define al ordenador por sus supuestas capacidades para "tratar" o "procesar" automáticamente la información. De acuerdo con lo visto más arriba, debe entenderse que este tratamiento o procesamiento de la información sólo tiene sentido para nosotros, que decidimos que tal "tira" o "cadena" de símbolos significa tal cosa, es decir, que *codificamos la información en cadenas de símbolos*; el ordenador se limita a manipular esas cadenas, dando normalmente como resultado otras cadenas que nosotros decodificamos.

Y hablando ya en términos generales, podemos considerar a un autómata como un dispositivo que manipula cadenas de símbolos que se le presentan a su entrada, produciendo otras tiras o cadenas de símbolos como salida. En el apartado 3.1 del capítulo 2 formalizaremos matemáticamente esta definición.

Un ordenador es un ejemplo de autómata, pero también son autómatas dispositivos más sencillos, como sumadores, contadores, etc., que veremos como ejemplos en el capítulo 2, o las mismas partes constituyentes de un ordenador: la unidad aritmética-lógica o la unidad de control son autómatas. Por otra parte, existen autómatas más complejos que un ordenador, como los robots controlados por ordenador. También puede estudiarse como autómatas determinadas funciones de los seres vivos, e incluso complejos sistemas ecológicos y socio-económicos.

2. Autómatas y máquinas secuenciales.

Concepto de estado

El autómata recibe los símbolos de entrada uno detrás de otro, distribuidos en el tiempo, es decir, secuencialmente. Además, en general, el símbolo de salida que en un instante determinado produce este autómata no sólo depende del último símbolo recibido a la entrada, sino de toda la secuencia o cadena, distribuida en el tiempo, que ha recibido hasta ese instante. Quiere esto decir que un autómata es una máquina secuencial, en el sentido de que opera sobre secuencias de símbolos, "recordando" en todo instante la "historia" de símbolos llegados hasta ese instante. Esto le diferencia de una máquina puramente combinatoria como sería, por ejemplo, un circuito lógico de los estudiados en el tema "Lógica".

Así pues, ante un determinado símbolo de entrada un autómata puede producir diferentes símbolos de salida, dependiendo de la historia o secuencia de todos los símbolos de entrada anteriores.

Obsérvese que hasta ahora venimos hablando de un autómata como una "caja negra" con una entrada en la que recibe símbolos y una salida, sobre la que deposita otros símbolos. Otra manera de enfocar el estudio de los autómatas es considerando lo que hay dentro de la caja negra (aunque sea de una manera abstracta, es decir, prescindiendo de la naturaleza de los componentes físicos y atendiendo sólo a las transformaciones de símbolos), y esto nos lleva a definir un concepto fundamental: El estado del autómata. *El estado es toda la información necesaria en un momento dado para poder deducir, dado un símbolo de entrada en ese momento, cual será el símbolo de salida.* Es decir, conocer el estado es lo mismo que conocer toda la historia de símbolos de entrada¹. Un autómata tendrá un determinado número de estados (en teoría, puede tener infinitos), y se encontrará en uno u otro según sea la historia de símbolos que le han llegado; si encontrándose en un estado determinado, recibe un símbolo también determinado, producirá un símbolo de salida y efectuará un cambio o *transición* a otro estado (también puede quedarse en el mismo). Estas ideas son fáciles de formalizar matemáticamente a partir de los conceptos de conjunto y función, y conducen a la definición de autómata que desarrollaremos en el capítulo siguiente.

3. Autómatas y lenguajes

Un campo importante dentro de la Informática, al que dedicaremos el último tema, está constituido por el estudio de los lenguajes y las gramáticas que los

¹ Realmente, no basta con conocer toda la historia de símbolos de entrada para saber cuál es la salida; es necesario conocer también el "estado inicial", es decir, el estado en el que se encontraba el autómata al recibir el primero de los símbolos de entrada.

generan. Los elementos de un lenguaje son sentencias, palabras, etc., formados a partir de un *alfabeto* (capítulo 1, apartado 5 del tema "Lógica"). Establecidas unas reglas gramaticales, una cadena de símbolos pertenecerá al correspondiente lenguaje si tal cadena se ha formado obedeciendo esas reglas; puede entonces pensarse en la posibilidad de construir un *autómata reconocedor* de ese lenguaje, tal que cuando reciba a su entrada una determinada secuencia de símbolos produzca, por ejemplo, un "1" a la salida si la secuencia es correcta, y un "0" si no lo es. De este modo, como veremos en su momento, a cada tipo de gramática corresponde un tipo de autómata.

4. Autómatas y álgebra

Las cadenas de entrada y salida de un autómata se forman a partir de los correspondientes alfabetos mediante una operación que consiste en poner los símbolos unos a continuación de otros. Esta operación se llama concatenación, y es asociativa. Por consiguiente, el conjunto de todas las cadenas con la concatenación tiene una estructura algebraica de semigrupo; si, además, definimos un elemento neutro, tendremos un monoide.

Por otra parte, como veremos en el capítulo siguiente, si el autómata es finito (es decir, si tiene un número finito de estados) puede determinarse un número finito de clases de equivalencia en el semigrupo (o monoide) de entrada, lo cual permite definir un semigrupo (o monoide) cociente llamado el semigrupo (o monoide) de la máquina, a partir del cual pueden formalizarse muchas cuestiones relativas al funcionamiento de los autómatas.

Siguiendo esta línea de trabajo, se ha elaborado en las últimas décadas una teoría abstracta de autómatas con una fuerte base algebraica que, según Arbib (1969), constituye "la matemática pura de la Informática".

5. Resumen

La teoría de autómatas, también llamada teoría algebraica de máquinas, permite estudiar de un modo sistemático las máquinas, más o menos complicadas, que realizan un procesamiento de la información y que actúan de manera discreta, es decir, la información se supone codificada a partir de un conjunto finito de símbolos que el autómata trata secuencialmente, uno detrás de otro. La teoría de autómatas proporciona métodos para el análisis y la síntesis de tales máquinas.

Los trabajos sobre lenguajes y gramáticas formales han evolucionado en una dirección que les ha conducido a encontrarse con la teoría de autómatas como herramienta matemática de gran utilidad.

La teoría de autómatas no sólo puede aplicarse a las "máquinas", en el sentido estricto que normalmente damos a esta palabra, sino también a muchos sistemas

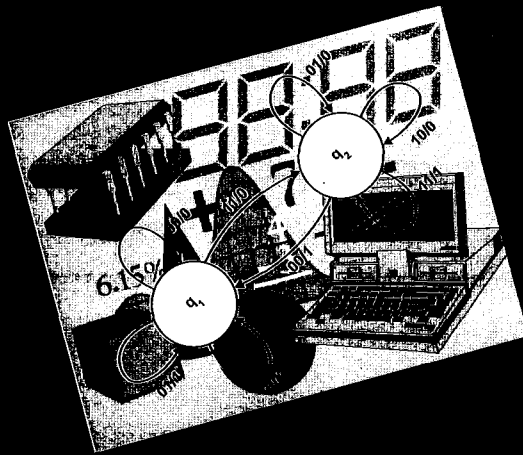
naturales, y, en general, permite estudiar procesos que dependen de una "historia", es decir, cuyo comportamiento presente es función del pasado.

En sus treinta años de historia la teoría de autómatas se ha constituido en una disciplina muy formalizada que sigue en evolución. Mientras la teoría básica (autómatas finitos deterministas) puede considerarse definitivamente establecida, se abren nuevas vías que actualmente son objeto de estudio de los investigadores y que ofrecen amplias perspectivas de aplicación: autómatas estocásticos, borrosos, adaptativos, de aprendizaje, etc.

Evidentemente, en este tema no podemos exponer ni siquiera resumir, toda la teoría de autómatas. Nuestro objetivo será presentar los principios básicos, desarrollando algunos ejemplos de aplicación para ver su utilidad práctica en diversos campos, especialmente el de la Informática.

Fundamentos de informática

2



2

Autómatas Finitos

1. Definición y representación de los autómatas

1.1. Definición

Un *autómata* es una quintupla:

$$A = \langle E, S, Q, f, g \rangle \quad [1.1.1.]$$

donde:

- E es un conjunto finito, llamado *conjunto de entradas* o *alfabeto de entradas*, cuyos elementos llamaremos *entradas* o *símbolos de entrada*.
- S es un conjunto finito, llamado *conjunto de salidas* o *alfabeto de salida*, cuyos elementos llamaremos *salidas* o *símbolos de salida*.
- Q es un conjunto llamado *conjunto de estados*.

- f es una función $f: E \times Q \rightarrow Q$, llamada *función de transición* o *función de estado siguiente*.
- g es una función $g: E \times Q \rightarrow S$, llamada *función de salida*.

Esta definición formal puede interpretarse como la descripción matemática de una máquina que, si en el instante t recibe una entrada $e \in E$ y se encuentra en el estado $q \in Q$, entonces da una salida $g(e, q)$, y pasa al estado $f(e, q)$ en el instante $t + 1$. (Suponemos una escala discreta de tiempos arbitraria: $t = 1, 2, 3, \dots$). Expresando de una manera explícita el tiempo, y si llamamos s a un elemento genérico de S , podemos escribir:

$$q(t+1) = f[e(t), q(t)] ; s(t) = g[e(t), q(t)]$$

A es un autómata finito si Q es un conjunto finito. En lo sucesivo, y hasta el capítulo 5, trataremos sólo con autómatas finitos, y abreviaremos escribiendo "AF".

1.2. Representación

1.2.1. Tabla de transiciones

Las funciones f y g pueden representarse mediante una tabla con tantas filas como estados y tantas columnas como entradas. Si la fila i corresponde al estado q_i y la columna j corresponde a la entrada e_j , en la intersección de ambas se escribirá $f(e_j, q_i)/g(e_j, q_i)$. Por ejemplo, sea el AF definido por los conjuntos

$$E = \{a, b\}$$

$$S = \{0, 1\}$$

$$Q = \{q_1, q_2, q_3\}$$

y las funciones de estado y de salida

$$f(a, q_1) = q_1; g(a, q_1) = 0$$

$$f(b, q_1) = q_2; g(b, q_1) = 1$$

$$f(a, q_2) = q_3; g(a, q_2) = 0$$

$$f(b, q_2) = q_2; g(b, q_2) = 0$$

$$f(a, q_3) = q_3; g(a, q_3) = 1$$

$$f(b, q_3) = q_1; g(b, q_3) = 0$$

En lugar de esto, es más cómodo representar f y g por la tabla de transiciones de la figura 2.1.

$\begin{array}{c} e \\ \backslash \\ q \end{array}$	a	b
q_1	$q_1/0$	$q_2/1$
q_2	$q_3/0$	$q_2/0$
q_3	$q_3/1$	$q_1/0$

Figura 2.1.

1.2.2. Diagrama de Moore

Otra forma de representar las funciones f y g es mediante un grafo orientado en el que cada nodo corresponde a un estado, y si $f(e, q_i) = q_j$ y $g(e, q_i) = s$, existe un arco dirigido del nodo correspondiente a q_i al correspondiente a q_j , sobre el que pondremos la etiqueta e/s . Por ejemplo, el AF definido por la tabla anterior puede representarse por el grafo de la figura 2.2. Este grafo suele llamarse diagrama de transiciones o diagrama de Moore.

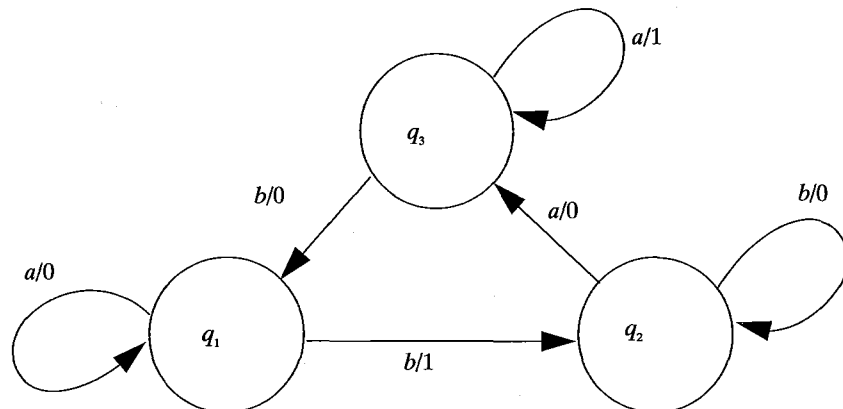


Figura 2.2.

1.3. Máquinas de Moore y de Mealy

El modelo general de autómatas que hemos definido se llama *máquina de Mealy*. Las funciones f y g determinan la salida y el estado siguiente cuando la máquina se encuentra en un estado $q \in Q$ y recibe una entrada $e \in E$. Ahora bien, por conveniencia matemática, es interesante considerar, además de los símbolos o elementos de E , un elemento neutro, λ ; físicamente, el decir que la entrada es λ , es lo mismo que decir que no hay ninguna entrada. Es inmediato entonces plantearse la siguiente pregunta: ¿qué ocurre si, estando un autómata en el estado $q \in Q$, recibe como entrada λ ? Para responder a esto, matemáticamente, habría que ampliar el dominio de f , que es $E \times Q$, a $\{E \cup \{\lambda\}\} \times Q$, y lo mismo el dominio de g . La ampliación del dominio de f no plantea ningún problema: se puede convenir que $f(\lambda, q) = q$ (es decir, físicamente, que si no hay entrada no se cambia de estado). Pero no ocurre lo mismo con g ; y ello se ve fácilmente si nos referimos al ejemplo desarrollado más arriba (figura 2.1): si llegamos a q_1 , ya sea de q_3 (por efecto de entrada b) o de q_1 (por a) la salida es 0, por lo que podemos asociar la salida 0 al estado q_1 y decir $g(\lambda, q_1) = 0$; sin embargo, no podemos definir $g(\lambda, q_2)$, ya que si llegamos a q_2 desde q_1 la salida es 1, mientras que si llegamos desde el propio q_2 la salida es 0. Es evidente que, en general, sólo puede definirse $g(\lambda, q)$ en el caso en que se cumpla que

$$[q = f(e_1, q_1) = f(e_2, q_2)] \rightarrow [g(e_1, q_1) = g(e_2, q_2)] \quad [1.3.1.]$$

es decir, que a q se le pueda asociar una salida y una sola. Si esto ocurre para todo $q \in Q$ podemos definir una función inyectiva $h: Q \rightarrow S$ tal que $g(e, q) = h[f(e, q)]$, $e \in \{E \cup \{\lambda\}\}$, $q \in Q$. En este caso, podemos decir que la salida sólo depende del estado, y el autómata se llama *máquina de Moore*. Expresando el tiempo de manera explícita:

$$s(t) = g[e(t), q(t)] = h[q(t)] = h[f[e(t-1), q(t-1)]]$$

En una máquina de Mealy las salidas están asociadas a las transiciones, mientras que en una máquina de Moore las salidas están asociadas a los estados, o, lo que es lo mismo, todas las transiciones que conducen a un mismo estado tienen asociada la misma salida. También podemos decir que una máquina de Mealy, en el instante de efectuar una transición necesita conocer una entrada $e \in E$ (ya que, en general, $g(\lambda, q)$ no está definida), mientras que en una máquina de Moore la entrada puede ser $e \in E$ o $e = \lambda$.

Puesto que toda máquina de Moore es una máquina de Mealy que cumple la condición [1.3.1] para todo $q \in Q$, parece en principio que las primeras son un subconjunto de las segundas. Sin embargo, vamos a demostrar que, dada una máquina de Mealy, siempre podremos encontrar una máquina de Moore equivalente (normalmente, a costa de aumentar el número de estados). En efecto, si tenemos una máquina de Mealy

$$A = \langle E, S, Q, f, g \rangle$$

siempre podemos definir un nuevo autómata

$$\hat{A} = \langle E, S, \hat{Q}, \hat{f}, \hat{g} \rangle$$

en el que \hat{Q} se obtiene escindiendo cada $q \in Q$ en tantos estados¹ q^s como salidas s puedan asociarse a q :

$$\hat{Q} = \{q^s \mid \exists (q' \in Q \text{ y } e \in E) \text{ tales que } f(e, q') = q, \text{ y } g(e, q') = s\}$$

y en el que \hat{f} y \hat{g} se definen así:

$$\hat{f}(e, q^s) = [f(e, q)]^{s(e, q)}$$

$$\hat{g}(e, q^s) = g(e, q)$$

De este modo, a cada $q^s \in \hat{Q}$ se le puede asociar una sola salida, s , y así tendremos una función de salida $\hat{h}: \hat{Q} \rightarrow S$ tal que $\hat{g}(e, q^s) = \hat{h}[\hat{f}(e, q^s)]$, por lo que \hat{A} será una máquina de Moore.

Concretemos estas ideas con un ejemplo. Tomemos el autómata cuyo diagrama es el de la figura 2.2. Como ya hemos visto, con q_1 siempre se puede asociar la salida 0; sin embargo, q_2 lo escindiremos en q_2^0 y q_2^1 , ya que la salida es 0 ó 1, según que vengamos de q_2 o de q_1 y, del mismo modo, escindiremos q_3 en q_3^0 y q_3^1 . De acuerdo con esto, y teniendo en cuenta las definiciones de f y g obtenemos el diagrama de la figura 2.3.

¹s aquí es un superíndice, no un exponente.

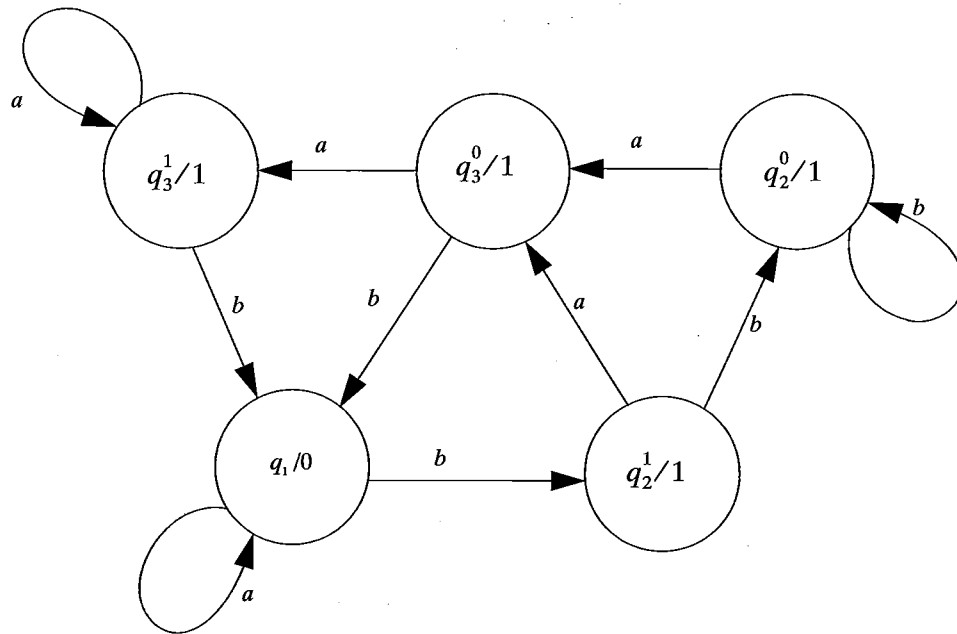


Figura 2.3.

Obsérvese que, al estar las salidas asociadas con los estados, todas las transiciones que conducen a un estado producen la misma salida, por lo que en lugar de rotular las salidas sobre los arcos las hemos incluido en los nodos. Del mismo modo, en la tabla de transiciones podemos incluir las salidas en la misma columna de estados; la tabla de esta máquina de Moore es entonces la de la figura 2.4.

En una máquina de Moore podemos considerar el monoide $\langle E^*, \cdot \rangle$, al que se llama *monoide libre de entrada*. En una máquina de Mealy sólo podemos hablar del *semigrupo libre de entrada*, $\langle E^+, \cdot \rangle$. En lo sucesivo siempre que hablemos de un autómata supondremos, a menos que se diga lo contrario, que se trata de una máquina de Moore, es decir, representaremos indistintamente la salida por la función de salida $g(e, q)$ o por $h(q)$ teniendo en cuenta que $g = h \circ f$.

$\begin{array}{c} e \\ q/s \end{array}$	a	b
$q_1/0$	q_1	q_2^1
$q_2^0/0$	q_3^0	q_2^0
$q_2^1/1$	q_3^0	q_2^0
$q_3^0/0$	q_3^1	q_1
$q_3^1/1$	q_3^1	q_1

Figura 2.4.

2. Ejemplos de autómatas como modelos

2.1. Detector de paridad

Un procedimiento sencillo y muy utilizado para detectar errores en una transmisión digital (por ejemplo, en una transferencia de datos de un periférico remoto al bus de datos), consiste en enviar un bit de paridad. Este bit puede ser tal que haga par el número total de "unos" enviados (paridad par), o que lo haga impar (paridad impar). Por ejemplo, supongamos que el periférico envía caracteres codificados en código ASCII de 8 bits con paridad par (es decir, el código es ASCII de 7 bits, y el octavo bit es el de paridad). El carácter "A", en ASCII de 7 bits, se codifica 1000001; luego en 8 bits será 01000001 (el bit de paridad se hace 0 para que el número total de "unos" sea par). Por el contrario, el código de "C" es 1000011, por lo que el bit de paridad deberá ser 1 y por consiguiente en 8 bits será 11000011.

En el punto emisor deberá existir un *generador de paridad*, y en el receptor, un *detector de paridad*. Este detector deberá dar una señal de error en el caso de que la paridad recibida no sea correcta, cosa que ocurrirá cuando en la transmisión haya habido una alteración en un bit (o en un número impar de bits). Si la paridad es correcta no dará error. (Obsérvese que si hay un número par de alteraciones en la transmisión este sistema no detecta el error, pero la probabilidad de que ocurra más de una alteración es muy pequeña. Existen, desde luego, otros procedimientos mejores de detección e incluso corrección de errores).

El alfabeto de entrada del detector es, evidentemente, $E = \{0, 1\}$. El alfabeto de salida constará de dos elementos ("error" y "no error"); podemos tomar el convenio de que sea también $S = \{0, 1\}$, donde "0" significa "no error" y "1" significa "error". El conjunto de estados puede ser $Q = \{q_0, q_1, q_2\}$, donde q_0 es el estado inicial, del que sólo se sale al recibir el primer bit; en q_1 se estará si se ha recibido un número par de bits y en q_2 si se ha recibido un número impar, de manera que, al finalizar la transmisión, si la máquina se ha quedado en q_1 es que no ha habido error ($s = 0$), y si se ha quedado en q_2 es que sí lo ha habido ($s = 1$), de acuerdo con esto, es fácil establecer el diagrama de Moore de la figura 2.5.

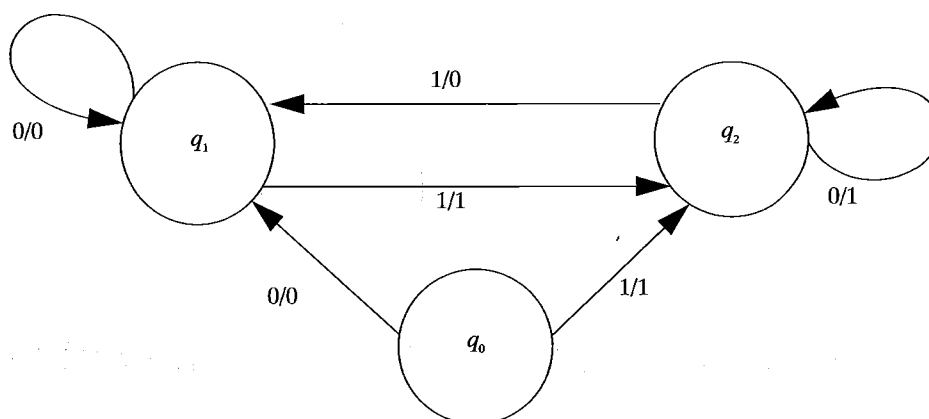


Figura 2.5.

Ahora bien, el estado q_0 puede fundirse con el q_1 . Ello equivale a convenir en que inicialmente, cuando no se ha recibido ningún bit (es decir, cuando se ha recibido λ) la salida es 0. Obtenemos así el diagrama de Moore de la figura 2.6, en el que cada estado tiene una salida, y sólo una, asociada (es una máquina de Moore).

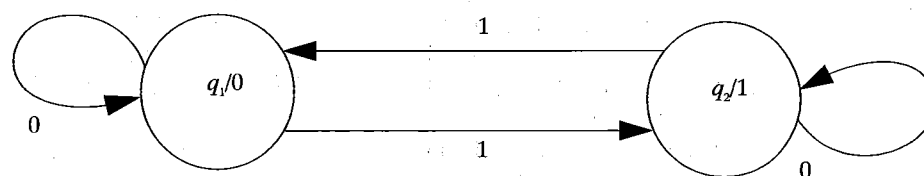


Figura 2.6.

2.2. Sumador binario serial

Un sumador binario es un dispositivo que suma números codificados en forma binaria y da el resultado también en binario. En el sumador serial los bits de los sumandos se presentan secuencialmente y por parejas, es decir, primero se pre-

sentan los dos bits de menor peso, el sumador los suma y obtiene el bit de menor peso del resultado (y toma nota del arrastre, si lo hay), luego los siguientes, etc. (figura 2.7).

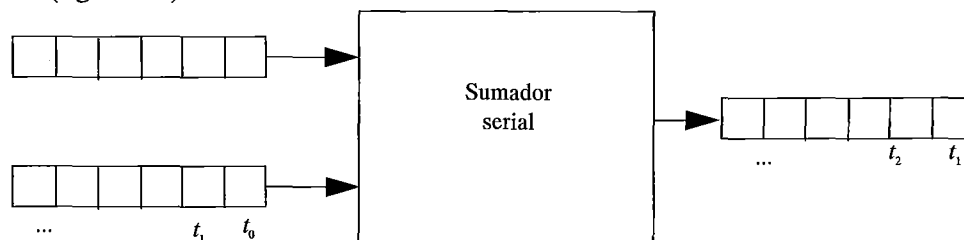


Figura 2.7.

Evidentemente, el sumador serial es un autómata, puesto que, en todo momento debe recordar si ha habido arrastre de los bits sumados anteriormente, es decir, la salida no sólo depende de la entrada actual, sino también de las anteriores. También es fácil ver que sólo necesita dos estados. En efecto, en cada momento, para efectuar una suma de dos bits, sólo hay dos posibles situaciones a considerar: que no exista arrastre de los anteriores o que sí lo haya; llamaremos q_1 y q_2 , respectivamente, a los estados correspondientes a esas situaciones.

Tenemos, por tanto:

$$E = \{00, 01, 10, 11\}$$

$$S = \{0, 1\}$$

$$Q = \{q_1, q_2\}$$

Inicialmente, el autómata estará en el estado q_1 (al recibir la primera pareja de bits no tiene que considerar ningún arrastre anterior). Si la primera pareja es 00, la salida deberá ser 0, y, como no hay arrastre, se quedará en q_1 ; si es 01 ó 10 deberá dar salida 1 también quedarse en q_1 , pero si recibe 11 la salida deberá ser 0, y, habrá arrastre, por lo que pasará a q_2 . Estando en q_2 , si recibe 00, como hay arrastre de la suma anterior, deberá dar como salida 1 pero ya no habrá arrastre para la suma siguiente, por lo que pasará a q_1 ; sin embargo, en cualquier otro caso (01, 10, 11) se quedará en q_2 , ya que sigue existiendo arrastre. Toda esta descripción se puede expresar con mayor concisión y claridad con el diagrama de Moore de la figura 2.8. Este AF es una máquina de Mealy, puesto que tanto q_1 como q_2 tienen asociadas las salidas 0 y 1. La máquina de Moore equivalente puede encontrarse siguiendo el procedimiento expuesto en el apartado 1.3, y resulta ser la descrita por el diagrama de la figura 2.9.

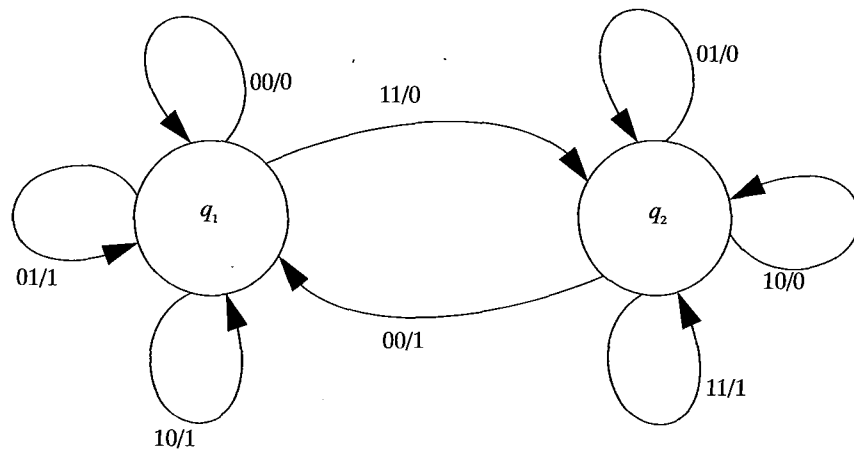


Figura 2.8.

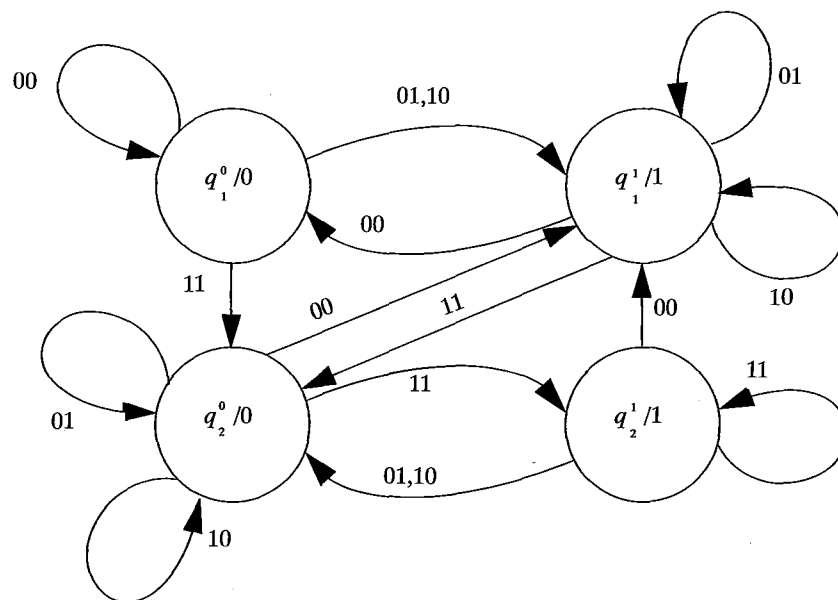


Figura 2.9.

2.3. El castillo encantado

El siguiente ejemplo, tomado de Ashby (1956), nos servirá para ilustrar cómo la teoría de autómatas tiene un campo de aplicación muy extenso: todo lo que se refiera a sistemas (en el más amplio sentido de la palabra) discretos con memoria; en este caso particular veremos cómo permite formalizar y resolver un problema de lógica en el que interviene el tiempo.

El problema es el expuesto en esta carta:

"Querido amigo: Al poco tiempo de comprar esta vieja mansión tuve la desagradable sorpresa de comprobar que está hechizada con dos sonidos de ultratumba que la hacen prácticamente inhabitable: un canto picaresco y una risa sardónica.

Aún conservo, sin embargo, cierta esperanza, pues la experiencia me ha demostrado que su comportamiento obedece a ciertas leyes, oscuras pero infalibles, y que puede modificarse tocando el órgano y quemando incienso.

En cada minuto, cada sonido está presente o ausente. Lo que cada uno de ellos hará en el minuto siguiente depende de lo que pasa en el minuto actual, de la siguiente manera:

El canto conservará el mismo estado (presente o ausente) salvo si durante el minuto actual no se oye la risa y toco el órgano, en cuyo caso el canto toma el estado opuesto.

En cuanto a la risa, si no quemó incienso, se oirá o no según que el canto esté presente o ausente (de modo que la risa imita al canto con un minuto de retardo). Ahora bien, si quemó incienso la risa hará justamente lo contrario de lo que hacía el canto.

En el momento en que le escribo estoy oyendo a la vez la risa y el canto. Le quedaré muy agradecido si me dice qué manipulaciones de órgano e incienso debo seguir para restablecer definitivamente la calma".

La carta, especialmente en su tercer párrafo, nos describe un sistema lógico secuencial que puede ser formalizado como un autómata finito. Existen dos variables de entrada (órgano e incienso), y como cada una de ellas dispone de dos valores posibles, tendremos por tanto cuatro entradas diferentes a las que llamaremos e_0 , e_1 , e_2 y e_3 .

e_0 : no tocar el órgano ni quemar incienso;

e_1 : no tocar el órgano pero quemar incienso;

e_2 : tocar el órgano pero no quemar incienso;

e_3 : tocar el órgano y quemar incienso.

También son cuatro los estados posibles:

q_0 : ni risa, ni canto;

q_1 : no risa, sí canto;

q_2 : sí risa, no canto;

q_3 : risa y canto.

En cuanto la salida, podemos considerar dos situaciones:

1: que haya algún sonido (salida asociada a los estados q_1, q_2, q_3);

2: que no haya ningún sonido (salida asociada al estado q_0).

Con esta nomenclatura, el problema se puede expresar diciendo que nos encontramos en un estado inicial, el q_0 , queremos pasar a un estado final, el q_0 , y se trata de encontrar la secuencia de entrada adecuada.

Siguiendo el enunciado, podemos obtener la tabla y el diagrama de transiciones, pero ello resulta mucho más fácil si utilizamos un formalismo lógico. Designemos por I y O unas variables booleanas que representen el incienso y el órgano, respectivamente (es decir, $I = 0$ si no se quema incienso, $I = 1$ si se quema, etc.) y por R y C otras variables que representen la risa y el canto ($R = 0$ si no se oye la risa, etc.). Tenemos una correspondencia inmediata entre los valores de estas variables y los conjuntos de entradas y estados definidos más arriba:

e	O	I	q	R	C
e_0	0	0	q_0	0	0
e_1	0	1	q_1	0	1
e_2	1	0	q_2	1	0
e_3	1	1	q_3	1	1

Si en el minuto t los valores de estas variables son O_t, I_t, C_t, R_t , en el minuto $t+1$ tomarán los valores dados por las siguientes expresiones lógicas, que no son más que otra forma de expresar los párrafos 4 y 5 de la carta:

$$(1) O_t \cdot \bar{R}_t = 1 \rightarrow C_{t+1} = \bar{C}_t$$

$$(2) O_t \cdot \bar{R}_t = 0 \rightarrow C_{t+1} = C_t$$

$$(3) I_t = 0 \rightarrow R_{t+1} = C_t$$

$$(4) I_t = 1 \rightarrow R_{t+1} = \bar{C}_t$$

De (1) y (2) se deduce que

$$(5) \ C_{t+1} = (O_t \cdot R_t) \oplus C_t$$

y de (3) y (4)

$$(6) \ R_{t+1} = I_t \oplus C_t$$

De (5) y (6) se pueden sacar inmediatamente las tablas de verdad de C_{t+1} y R_{t+1} en función de O_t, I_t, C_t, R_t :

O_t	I_t	R_t	C_t	R_{t+1}	C_{t+1}
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	0	1

Volviendo a la correspondencia establecida entre O, I y e , y entre C, R y q , la anterior tabla de verdad nos conduce a la tabla de transiciones de la figura 2.10:

$q \backslash e$	e_0	e_1	e_2	e_3
$q_0/0$	q_0	q_2	q_1	q_3
$q_1/1$	q_3	q_1	q_2	q_0
$q_2/1$	q_0	q_2	q_0	q_2
$q_3/1$	q_3	q_1	q_3	q_1

Figura 2.10.

Y de aquí podemos dibujar el diagrama de Moore de la figura 2.11.

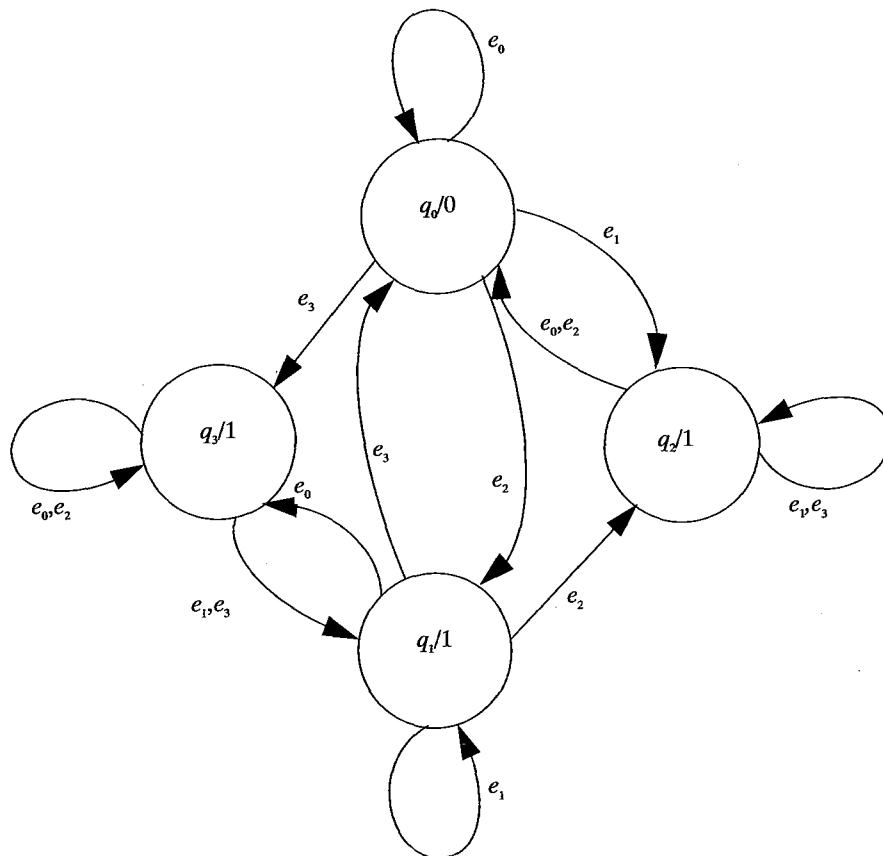


Figura 2.11.

A la vista de este diagrama, la solución al problema aparece fácilmente: pasar primero de q_3 a q_1 mediante e_1 ó e_3 , y luego a q_0 mediante e_3 ; o bien, respondiendo en los mismos términos epistolares, *"durante un minuto, queme usted incienso (tocando o no el órgano, es indiferente), y desaparecerá la risa; durante el minuto siguiente, queme incienso y toque el órgano, y al finalizar ese minuto cese toda actividad, con lo que, si no vuelve a manipular ni el órgano ni el incienso, se habrá librado para siempre de tan molestos moradores"*. Hay desde luego, otras soluciones, pero todas ellas con secuencias de entrada más largas; por ejemplo, de q_1 puede pasarse a q_2 con e_2 , y de aquí a q_0 con e_0 ó e_2 . Obsérvese que, afortunadamente para el propietario de la casa, el estado q_0 es estable, en el sentido de que con la entrada e_0 (es decir, $I = 0$, $O = 0$) el siguiente estado es el mismo q_0 .

3. Comportamiento de un autómata

3.1. Otra definición de autómata

En el capítulo 1 comenzamos hablando de los autómatas como dispositivos que producen cadenas de símbolos a la salida en respuesta a cadenas de símbolos presentadas a la entrada. Según esto, podríamos definir un autómata como una función:²

$$F^*: E^* \rightarrow S^* \quad [3.1.1.]$$

que hace corresponder a cada cadena de entrada, $x \in E^*$, una cadena de salida, $F^*(x) = y \in S^*$.

Ahora bien, vamos a ver que el autómata queda perfectamente definido restringiendo el rango de la función de S^* a S , es decir, podemos definir un autómata como una función

$$F: E^* \rightarrow S \quad [3.1.2.]$$

que hace corresponder a cada cadena de entrada, $x \in E^*$, el último símbolo obtenido como salida, $F(x) = s \in S$. En efecto, si $x = e_0 e_1 \dots e_{n-1}$, tendremos como símbolos de salida

$F(e_0)$ en el instante 1

$F(e_0 e_1)$ en el instante 2

²Utilizamos la notación explicada en el apartado 5 del capítulo 1 del Tema "Lógica".

$$F(e_0 e_1 \dots e_{n-1}) = F(x) \text{ en el instante } n$$

Por consiguiente, la cadena de salida $F^*(x)$ se obtendrá concatenando todos estos símbolos:

$$F^*(x) = F^*(e_0 e_1 \dots e_{n-1}) = F(e_0) F(e_0 e_1) \dots F(e_0 e_1 \dots e_{n-1}),$$

lo que nos muestra que F^* queda determinada conociendo F .

Consideremos, por ejemplo, el autómata sumador binario serial estudiado en el apartado 2.2, y supongámoslo efectuando la suma $010110 + 011011 = 110001$. En el instante t_0 recibirá por la entrada los bits de menor peso, es decir, $e_0 = 01$; en el instante t_1 tendremos $e_1 = 11$, etc. Así, la cadena de entrada será:

$$x = 01.11.10.01.11.00$$

(Obsérvese que, en contra de lo que es habitual, utilizamos un punto para indicar la concatenación, a fin de evitar ambigüedades en este caso, ya que los símbolos de entrada están formados por dos símbolos de nuestro alfabeto ordinario. Obsérvese también que las cadenas se escriben de izquierda a derecha en el tiempo, con lo que resulta un orden de escritura inverso al habitual en aritmética).

En el instante t_0 tendremos como salida:

$$F(e_0) = F(01) = 1;$$

$$\text{en } t_1: F(e_0 e_1) = F(01.11) = 0;$$

$$\text{en } t_2: F(e_0 e_1 e_2) = F(01.11.10) = 0;$$

$$\text{en } t_3: F(e_0 e_1 e_2 e_3) = F(01.11.10.01) = 0;$$

$$\text{en } t_4: F(e_0 e_1 e_2 e_3 e_4) = F(01.11.10.01.11) = 1;$$

$$\text{y en } t_5: F(e_0 e_1 e_2 e_3 e_4 e_5) = F(01.11.10.01.11.00) = 1;$$

De modo que la cadena total de salida es:

$$F^*(x) = F(e_0) F(e_0 e_1) \dots F(e_0 e_1 \dots e_5) = 100011,$$

que es el resultado de la suma escrito de izquierda a derecha según se van obteniendo los bits del resultado a partir del menor peso.

La definición [3.1.2] considera al autómata exclusivamente desde el punto de vista de entrada-salida, es decir, como una "caja negra", a diferencia de la definición [1.1.1], en la que se contempla lo que sucede en el interior de la "caja". Algunos autores, para resaltar la diferencia entre ambos, les dan distintos nombres, y así, por ejemplo, al autómata definido por [3.1.2] le llaman "máquina", y al definido según [1.1.1], "circuito"; y este mismo convenio seguiremos nosotros en adelante cuando nos interese destacar que nos referimos a una u otra definición.

En este punto, es natural que surjan dos preguntas inmediatamente: dada una máquina, ¿podemos encontrar su circuito? Y, evidentemente, la inversa. Para responder a ellas se hace precisa una consideración matemática previa sobre la definición [1.1.1]. En efecto, el dominio de las funciones f y g es $E \times Q$, lo que quiere decir que estas funciones nos permiten obtener el estado siguiente y la salida conociendo el estado actual y el *símbolo* de entrada. Para conocer la respuesta de circuito no a un símbolo, sino a una *cadena* de entrada es necesario ampliar el dominio a $E^* \times Q$.

3.2. Ampliación del dominio de las funciones de un autómata

En el apartado 1.3 vimos que el dominio de g podía ampliarse de $E \times Q$ a $\{E \cup \{\lambda\}\} \times Q$ solamente si existe una función de salida $h: Q \rightarrow S$; en este caso decíamos que el autómata es una máquina de Moore, y teníamos:

$$f(\lambda, q) = q; g(\lambda, q) = h[f(\lambda, q)] = h(q)$$

Para extender ahora el dominio a $E^* \times Q$ basta con definir, para todo $x_1, x_2 \in E^*$,

$$\begin{aligned} f(x_1 x_2, q) &= f[x_2, f(x_1, q)] \\ g(x_1 x_2, q) &= g[x_2, f(x_1, q)] = h[f(x_1 x_2, q)] \end{aligned}$$

3.3. El comportamiento de entrada-salida, o las máquinas definidas por un circuito

Definición 3.3.1. Dado un autómata (circuito) $A = \langle E, S, Q, f, g \rangle$, definimos el comportamiento de entrada-salida de A inicializado en el estado q por

$$C_q: E^* \rightarrow S$$

que aplica a cada $x \in E^*$ un $s = g(x, q)$.

Es decir, si en el instante t_0 el autómata se encuentra en el estado q e introducimos la cadena $x = e_1 e_2 \dots e_n$, se obtendrán las salidas

$$C_q(e_1) \text{ en } t = t_0$$

$$C_q(e_1 e_2) \text{ en } t = t_0 + 1$$

$$C_q(e_1 e_2 \dots e_n) = C_q(x) \text{ en } t = t_0 + n - 1$$

Vemos así que para todo autómata (circuito) pueden definirse, en principio, tantas funciones de la forma [3.1.2] (es decir, tantas "máquinas") como estados tenga el autómata, aunque hay que advertir que algunas de estas funciones pueden ser idénticas entre sí (lo que correspondería a estados equivalentes, según la definición que daremos enseguida).

3.4. Equivalencia y accesibilidad

Damos a continuación una serie de definiciones cuyo sentido se captará mejor con ayuda de ejemplos, que desarrollaremos en el apartado 3.5.

Definición 3.4.1. Dados dos autómatas con los mismos alfabetos de entrada y salida, $A_1 = \langle E, S, Q_1, f_1, g_1 \rangle$ y $A_2 = \langle E, S, Q_2, f_2, g_2 \rangle$, $q_1 \in Q_1$ es *equivalente* a $q_2 \in Q_2$ si $C_{q_1} = C_{q_2}$.

La misma definición sirve para estados equivalentes dentro de un mismo autómata: basta considerar que $Q_1 = Q_2$, $f_1 = f_2$, $g_1 = g_2$.

Definición 3.4.2. Un autómata está en *forma mínima* (o es *observable*) si

$$(C_{q_1} = C_{q_2}) \rightarrow (q_1 = q_2)$$

Es decir, en un autómata en forma mínima no existen estados equivalentes.

Definición 3.4.3. Los autómatas $A_1 = \langle E, S, Q_1, f_1, g_1 \rangle$ y $A_2 = \langle E, S, Q_2, f_2, g_2 \rangle$, son *equivalentes* si

$$\{C_{q_1} \mid q_1 \in Q_1\} = \{C_{q_2} \mid q_2 \in Q_2\}$$

Definición 3.4.4. q_2 es *accesible* desde q_1 si existe $x \in E^*$ tal que $f(x, q_1) = q_2$.

Definición 3.4.5. El *subautómata conectado* de un autómata $A = \langle E, S, Q, f, g \rangle$ es $A^c = \langle E, S, Q^c, f^c, g^c \rangle$, con

$$Q^c = \{q_2 \in Q \mid (\exists x \in E^*) (\exists q_1 \in Q) (f(x, q_1) = q_2)\}$$

y f^c y g^c son las restricciones de f y g de $E^* \times Q$ a $E^* \times Q^c$.

Es decir, el subautómata conectado a un autómata está formado por todos los estados del autómata original que son accesibles desde algún otro estado.

Definición 3.4.6. Un autómata A es *fuertemente conectado* si $A = A^c$.

3.5. Ejemplos

3.5.1. Autómata reconocedor de la cadena 010

Considérense los AF dados por los diagramas de las figuras 2.12 y 2.13.

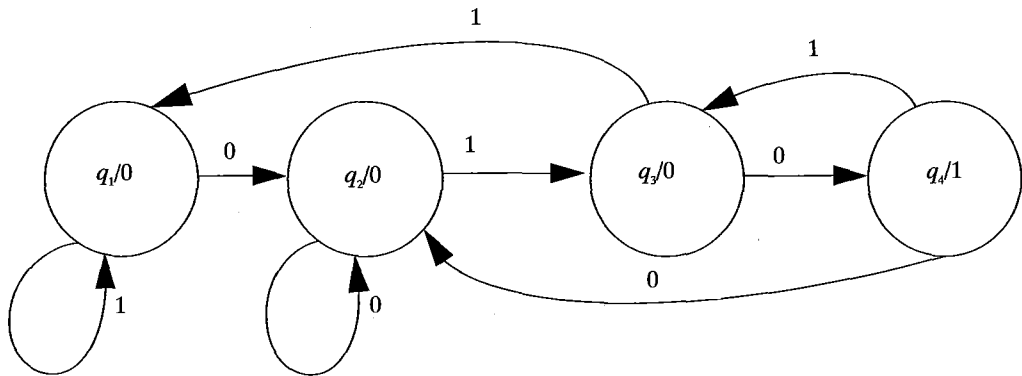


Figura 2.12.

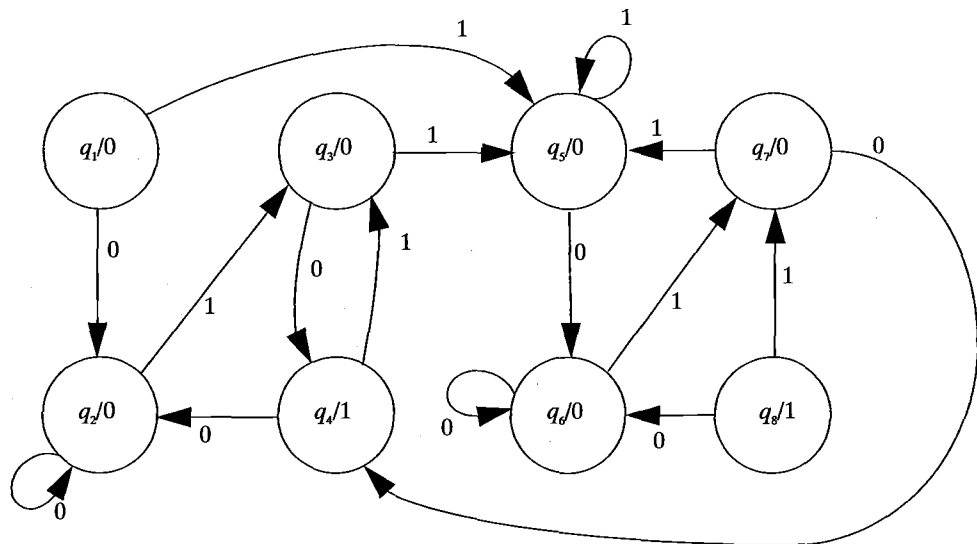


Figura 2.13.

- Calcular C_{q_1} , C_{q_2} , C_{q_3} , y C_{q_4} (en la figura 2.12) para cadenas de entrada de longitud igual o inferior a 3.
- Comprobar que el AF de la figura 2.12 está en forma mínima y es fuertemente conectado. ¿Ocurre lo mismo con el de la figura 2.13?

- c) Comprobar que ambos autómatas son equivalentes.

Pasemos a resolver las cuestiones planteadas.

- a) Basta con seguir, para cada estado inicial, las transiciones provocadas sucesivamente por cada símbolo de la cadena (leída ésta de izquierda a derecha) y anotar la salida final. Resumimos los resultados en la tabla de la figura 2.14.

	λ	0	1	00	01	10	11	000	001	010	011	100	101	110	111
C_{q_1}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
C_{q_2}	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
C_{q_3}	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
C_{q_4}	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0

Figura 2.14.

- b) Para comprobar que un AF está en forma mínima hay que ver si el comportamiento es distinto para cada estado. Esto no es fácil con los conocimientos que tenemos ahora, puesto que habría probar con diferentes cadenas de entrada hasta que se observe una diferencia de comportamiento entre dos estados para la misma cadena. Pero como E^* es infinito, si no encontramos tal diferencia después de un número finito de cadenas no podemos garantizar que el AF esté en forma reducida. (Empleando una terminología que se definirá con precisión en el tema siguiente, no tenemos un algoritmo). Ahora bien, en el apartado 5.3 demostraremos que basta con ensayar todas las cadenas x tales que $\lg(x) \leq n - 1$ (n = número de estados); y si $C_{q_1}(x) = C_{q_2}(x)$ para esas cadenas podremos asegurar que $C_{q_1}(x) = C_{q_2}(x) \forall x \in E^*$. De este modo tendremos un algoritmo, ya que el número necesario de ensayos es finito. En el autómata de la figura 2.12 se ve enseguida, con ayuda de la tabla de la figura 2.14, que

$$C_{q_1}(\lambda) = 0; C_{q_2}(\lambda) = 0; C_{q_3}(\lambda) = 0; C_{q_4}(\lambda) = 1$$

por lo que q_4 no es equivalente a ninguno de los otros tres. Observando los comportamientos para la entrada $x = 0$ deducimos que q_3 tampoco es equivalente a ninguno, y, de igual modo, la no equivalencia de q_2 la deducimos de la entrada $x = 10$.

Por consiguiente, podemos asegurar que el AF de la figura 2.12 está en forma mínima. No ocurre lo mismo con el de la figura 2.13. En efecto, si se van ensayando cadenas diferentes de entrada se irá viendo que $C_{q_1}(x) =$

$C_{q_5}(x)$, $C_{q_2}(x) = C_{q_6}(x)$, $C_{q_3}(x) = C_{q_7}(x)$ y $C_{q_4}(x) = C_{q_8}(x)$. Se puede comprobar que esto ocurre para todas las cadenas de longitud igual o inferior a 7 ($n = 8$), por lo que q_1 es equivalente a q_5 , q_2 a q_6 , q_3 a q_7 y q_4 a q_8 , y, por consiguiente, el AF no está en forma mínima.

El carácter de fuertemente conectado se ve por simple inspección de los diagramas. En efecto, en la figura 2.12 vemos que cualquiera de los cuatro estados tienen por lo menos un arco que entra en él, es decir, todos los estados son accesibles desde algún otro, y, por tanto, el autómata es fuertemente conectado. Sin embargo, en la figura 2.13 podemos observar que ni q_1 ni q_8 son accesibles desde ningún otro estado, por lo que el autómata no es fuertemente conectado.

- c) Ensayando todas las cadenas de longitud igual o inferior a 7 deducimos que q_1 y q_4 de la figura 2.13 son equivalentes a q_1 de la figura 2.12, q_2 y q_6 a q_2 , q_3 y q_7 a q_3 y q_4 y q_8 a q_4 . Por tanto, ambos AF son equivalentes.

Finalmente, justifiquemos el título puesto a este ejemplo, aunque los autómatas reconocedores serán objeto de estudio en capítulo 4. Puede comprobarse que siempre que en la cadena de entrada aparezca la sucesión 010 la salida del autómata es 1 al recibir el último símbolo de la sucesión (el segundo 0); en caso contrario la salida será 0, y no dará 1 hasta que en la cadena de entrada no vuelva a aparecer seguidos un 0, un 1 y luego otro 0.

3.5.2. Ejemplo 2: Equivalencia máquina de Moore-máquina de Mealy

En el apartado 1.3 decíamos que para cualquier máquina de Mealy se puede obtener una máquina de Moore equivalente, y construíamos esta máquina escindiendo cada estado de la primitiva en tantos estados como salidas pudieran asociársele. También decíamos que en lo sucesivo nos referiríamos siempre a máquinas de Moore, en las que podemos considerar el elemento neutro λ en la entrada, y, por consiguiente, el monoide libre de entrada, E^* , mientras que en el caso de máquinas de Mealy tendríamos que trabajar con el semigrupo libre de entrada, E^+ . Al definir la equivalencia entre autómatas (Definición 3.4.3) nos hemos basado en el comportamiento de entrada-salida (Definición 3.3.1), que se ha definido no con E^+ , sino con E^* , suponiendo implícitamente que los autómatas considerados son máquinas de Moore. Lo que pretendemos en este ejemplo es ver que la máquina de Moore construida a partir de una de Mealy como se indica en el apartado 1.3 es efectivamente equivalente a ella, en el sentido de equivalencia de comportamiento. Para ello tenemos que ver que los conjuntos de los comportamientos de ambas máquinas son idénticos, con la salvedad de que no tiene sentido hablar de la entrada λ ; es decir, por esta sola vez, digamos que el comportamiento para un estado q es

$$C_q: E^+ \rightarrow S$$

Pues bien, sobre el ejemplo de las figuras 2.2 y 2.3, y si llamamos A a la primera y \hat{A} a la segunda, podemos comprobar que

$$C_{q_1}(a) = \hat{C}_{q_1}(a) = 0; C_{q_1}(b) = \hat{C}_{q_1}(b) = 1; C_{q_1}(aa) = \hat{C}_{q_1}(aa) = 0;$$

$$C_{q_1}(ab) = \hat{C}_{q_1}(ab) = 1; C_{q_1}(ba) = \hat{C}_{q_1}(ba) = 0;$$

etc., es decir,

$$C_{q_1} = \hat{C}_{q_1}$$

$$\text{Análogamente se ve que } C_{q_2} = \hat{C}_{q_1}^0 = \hat{C}_{q_2}^1 \text{ y que } C_{q_3} = \hat{C}_{q_3}^0 = \hat{C}_{q_3}^1.$$

En general, lo que ocurre es que todos los estados q^* resultantes de una escisión de q tienen igual comportamiento (salvo, naturalmente, para $x = \lambda$), ya que se toma

$$\hat{g}(x, q^*) = g(x, q)$$

para todos los q^* .

4. Capacidad de respuesta de un autómata finito

4.1. Introducción

Hemos visto que un circuito con n estados puede realizar hasta n máquinas diferentes. Cada una de estas máquinas viene definida por el comportamiento de entrada-salida para el correspondiente estado.

El número de cadenas diferentes en E^* es infinito. Pero como el autómata es finito es imposible que responda de distinta manera a cada una de ellas. Es decir, debe ser posible particionar E^* en un número finito de subconjuntos tales que el autómata sea incapaz de distinguir dos cadenas pertenecientes al mismo subconjunto. Veremos que tales subconjuntos pueden considerarse como clases de equivalencia; naturalmente, cuanto mayor sea el número de clases de equivalencia mayor será la capacidad del autómata para responder de distinta manera a cadenas diferentes. Cuando decimos que el autómata "responde" a una cadena o "distingue" entre una u otra pesamos en el símbolo de salida asociado a cada cadena, de acuerdo con la definición de "máquina" (expresión [3.1.2]). Si consideramos exclusivamente máquinas de Moore, existirá una función de salida $h: Q \rightarrow S$. Vamos en ese apartado a simplificar el análisis, suponiendo que h es biyectiva, es

decir, que a cada estado podemos asignarle una salida diferente³. En este caso, dos cadenas pertenecerán a la misma clase de equivalencia (serán indistinguibles para todas las máquinas definidas por el circuito) si, considerando un estado inicial cualquiera, el estado final es el mismo para ambas cadenas. Nos vemos así conducidos a estudiar, para cada cadena, funciones de la forma $Q \rightarrow Q$, y, si llamamos Q^Q al conjunto de todas esas funciones, el comportamiento global del autómata vendrá determinado por una función $K: E^* \rightarrow Q^Q$, que asigna a cada cadena de entrada una función $Q \rightarrow Q$. Si el número de estados diferentes es n , habrá n^n funciones $Q \rightarrow Q$, por lo que el número máximo de clases de equivalencia en E^* será n^n .

Vamos a formalizar estas ideas, y para ello comenzaremos por recordar algunos conceptos de álgebra, pero antes debemos advertir al lector que este apartado, que es el más teórico del tema, no es imprescindible para la comprensión del resto, y, si lo que desea, puede omitirlo y saltar al apartado 5 (minimización de AF).

4.2. Repaso de algunos conceptos de álgebra

4.2.1. Monoide de transformaciones de un conjunto

Una transformación t en un conjunto C se define como una función de C en sí mismo: $t: C \rightarrow C$.

Teorema 4.2.1.1. Sea C un conjunto cualquiera y sea $C^C = \{t: C \rightarrow C\}$ el conjunto de todas las transformaciones en C . Entonces $\langle C^C, \circ \rangle$, donde " \circ " representa la ley de composición de funciones t , es un monoide, llamado *monoide de transformaciones* de C .

La demostración es casi inmediata, teniendo en cuenta la evidencia de que la composición de funciones t es una operación cerrada y asociativa, es decir, si $t_a, t_b, t_c \in C^C$,

$$t_a \circ t_b \in C^C$$

y

$$(\forall c \in C) [t_a \circ (t_b \circ t_c)((c))] = (t_a \circ t_b) \circ t_c(c) = t_a(t_b(t_c(c)))]$$

³Esta simplificación no resta generalidad al análisis. En efecto, lo que haremos será estudiar la "respuesta de estados" en el sentido de que para cada entrada nos fijaremos no en la salida, sino en las transiciones que provoca entre los estados. Si h no fuera biyectiva lo único que podría ocurrir es que dos cadenas diferentes en cuanto a su "respuestas de estados" fueran indistinguibles en cuanto a la salida, pero esto es fácil de analizar conociendo la función $h: Q \rightarrow S$.

Además, podemos definir un elemento neutro en C^C , $t_1: C \rightarrow C$, tal que $t_1 \circ t_i = t_i \circ t_1 = t_i$; este elemento neutro es $t_1(x) = x$. Por consiguiente, $\langle C^C, \circ \rangle$ cumple las condiciones para ser un monoide.

Ejemplo. Sea $C = \{0, 1\}$. Veamos cuál es el monoide de transformaciones de C . C^C tendrá cuatro elementos:

$$t_0: t_0(0) = 0, t_0(1) = 0$$

$$t_1: t_1(0) = 0, t_1(1) = 1$$

$$t_2: t_2(0) = 1, t_2(1) = 0$$

$$t_3: t_3(0) = 1, t_3(1) = 1$$

(Obsérvese que, acorde con la notación anterior, t_1 es el elemento neutro).

La composición de t_2 con t_3 , por ejemplo, será:

$$t_2 \circ t_3(0) = t_2(t_3(0)) = t_2(1) = 0$$

$$t_2 \circ t_3(1) = t_2(t_3(1)) = t_2(1) = 0$$

luego

$$t_2 \circ t_3 = t_0$$

Análogamente pueden hallarse las demás composiciones; el resultado puede ponerse en forma tabular (tabla del monoide):

\circ	t_0	t_1	t_2	t_3
t_0	t_0	t_0	t_0	t_0
t_1	t_0	t_1	t_2	t_3
t_2	t_3	t_2	t_1	t_0
t_3	t_3	t_3	t_3	t_3

Esta tabla representa la operación \circ en el conjunto C^C , y por tanto, describe al monoide $\langle C^C, \circ \rangle$.

4.2.2. Homomorfismo entre monoides

Definición 4.2.2.1. Si $\langle S_1, * \rangle$ y $\langle S_2, \cdot \rangle$ son dos semigrupos, una función $f: S_1 \rightarrow S_2$ decimos que es un *homomorfismo* entre ambos semigrupos si preserva la ley de composición interna, es decir, si

$$(\forall a, b \in S_1) [f(a * b) = f(a) \cdot f(b)]$$

o, expresado gráficamente, si el diagrama

$$\begin{array}{ccc} S_1 \times S_1 & \xrightarrow{*} & S_1 \\ f \times f \downarrow & & \downarrow f \\ S_2 \times S_2 & \xrightarrow{\cdot} & S_2 \end{array}$$

es conmutativo.

Definición 4.2.2.2. Un *isomorfismo* entre semigrupos es un homomorfismo en el que la función f es biyectiva.

Definición 4.2.2.3. Si S_1 es un monoide con elemento neutro e_1 , S_2 es un monoide con elemento neutro e_2 , f es un homomorfismo (isomorfismo) entre S_1 y S_2 , y se cumple que

$$f(e_1) = e_2$$

entonces f es un *homomorfismo (isomorfismo) monoide*.

Ejemplos:

- 1) Si $P = \{1, 2, 3, \dots\}$ y $N = \{0, 1, 2, 3, \dots\}$, entonces $\langle P, \cdot \rangle$, $\langle N, + \rangle$ y $\langle N, \cdot \rangle$ son monoides, mientras que $\langle P, + \rangle$ es un semigrupo, pero no monoide. (¿Por qué?). La multiplicación por un número natural a en $\langle N, + \rangle$, es decir, $\langle N, + \rangle \rightarrow \langle N, + \rangle: n \rightarrow a \cdot n$ es un homomorfismo, ya que $a \cdot (n_1 + n_2) = a \cdot n_1 + a \cdot n_2$. Sin embargo, en general, no es un homomorfismo para $\langle N, \cdot \rangle$. (¿En qué casos particulares lo es?).
- 2) Llamemos R al conjunto de los números reales. $\langle R, + \rangle$ y $\langle R, \cdot \rangle$, son monoides. (¿Por qué?). La función

$$\langle R, + \rangle \rightarrow \langle R, \cdot \rangle: r \rightarrow a^r \quad (a, r \in R)$$

es un homomorfismo monoide, ya que

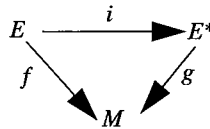
$$a^{r_1+r_2} = a^{r_1} \cdot a^{r_2} \text{ y } 0^r = 1$$

(¿es isomorfismo?).

4.2.3. Monoide libre y homomorfismo

Hemos definido en el apartado 5.2 del capítulo 1 del tema "lógica" el monoide libre generado por un alfabeto E , $\langle E^*, \cdot \rangle$.

Teorema 4.2.3.1. Sea $i: E \rightarrow E^*$ la función que aplica todo elemento de E en la correspondiente cadena de longitud unidad, es decir, $(\forall a \in E) (i(a) = a)$ y sea f cualquier función de E en el conjunto de cualquier monoide $\langle M, * \rangle$. Entonces, existe un único homomorfismo monoide $g: \langle E^*, \cdot \rangle \rightarrow \langle M, * \rangle$ tal que $g \circ i = f$, es decir, tal que el diagrama



es conmutativo.

Demostración:

Para cadenas de longitud 1 podemos definir $g(a) = f(a)$ para que se satisfaga $f(a) = g \circ i(a) = g(i(a)) = g(a)$. Si x es una cadena de longitud $l \geq 2$ podemos descomponerla en $x = ya_n$, donde $\lg(y) = l - 1$ y $\lg(a_n) = 1$, y tendremos:

$$g(x) = g(ya_n) = g(y) * g(a_n)$$

para que g sea un homomorfismo. Como $\lg(a_n) = 1$,

$$g(x) = g(y) * f(a_n)$$

Del mismo modo, podemos descomponer y en za_{n-1} , y así sucesivamente, con lo que, por inducción sobre la longitud de la cadena, si $x = a_1 a_2 \dots a_n$ podemos determinar $g(x)$ así:

$$g(x) = f(a_1) * f(a_2) \dots * f(a_n)$$

Finalmente, si e es el elemento neutro de $\langle M, * \rangle$, haremos $g(\lambda) = e$ para obtener un homomorfismo monoide.

Este teorema nos permite extender el dominio de cualquier función $E \rightarrow M$ de alfabeto E en el conjunto de un monoide $\langle M, * \rangle$ a un homomorfismo monoide $\langle E^*, > \rightarrow \langle M, * \rangle$, y nos será de utilidad más adelante.

4.2.4. Relaciones de congruencia y monoide cociente

Definición 4.2.4.1. Una *relación binaria* R en un conjunto C es un subconjunto de $C \times C$. Si c_1 y c_2 son elementos de C , decimos que c_1 y c_2 están relacionados por R si $(c_1, c_2) \in R$, y generalmente se escribe: $c_1 R c_2$.

Definición 4.2.4.2. R es una *relación de equivalencia* en un conjunto C si y sólo si R es una relación binaria que tiene las propiedades:

- Reflexiva: $(\forall c \in C) (c R c)$
- Simétrica: $(\forall c_1, c_2 \in C) [(c_1 R c_2) \rightarrow (c_2 R c_1)]$
- Transitiva: $(\forall c_1, c_2, c_3 \in C) [(c_1 R c_2) \wedge (c_2 R c_3) \rightarrow (c_1 R c_3)]$

Definición 4.2.4.3. Dados un conjunto C y una relación de equivalencia R , la *clase de equivalencia* de $c \in C$ módulo R , es $[c]_R = \{x \mid x R c\}$. (En adelante, siempre que no se preste a ambigüedad, suprimimos el subíndice R .) El *conjunto cociente* C/R es el conjunto de las clases de equivalencia de C módulo R . El número de elementos de C/R es el *índice* de la equivalencia.

Así, una relación de equivalencia origina una división de C en subconjuntos disjuntos, es decir, una partición.

Definición 4.2.4.4. Dado un monoide $\langle M, * \rangle$ y una relación de equivalencia, R , en M , R es una *relación de congruencia* en $\langle M, * \rangle$ si $a R b$ implica que $(a * c) R (b * c)$ y $(c * a) R (c * b)$ para todo $c \in M$.

Teorema 4.2.4.5. Si R es una relación de congruencia en el monoide $\langle M, * \rangle$, el conjunto cociente $M/R = \{[a] \mid a \in M\}$ con la operación " \cdot " definida por

$$[a] \cdot [b] = [a * b]$$

es un monoide.

Demostración:

En primer lugar hay que demostrar que la operación " \cdot " está bien definida sobre las clases de equivalencia, es decir, que el resultado es independiente de que se tome un miembro u otro de la clase. Para ello, sean $a_1 \in [a]$, $a_2 \in [a]$, $b_1 \in [b]$, $b_2 \in [b]$. Tenemos pues que $a_1 R a_2$ y $b_1 R b_2$, y, como R es una relación de congruencia, podemos escribir:

$$(a_1 * b_1) R (a_1 * b_2) \text{ y } (a_1 * b_2) R (a_2 * b_2)$$

y, por la propiedad transitiva de R ,

$$(a_1 * b_1) R (a_2 * b_2)$$

es decir,

$$[a_1 * b_1] = [a_2 * b_2],$$

lo que indica que $[a] \cdot [b]$ está bien definida.

Para ver que $\langle M/R, \cdot \rangle$ es un monoide, basta con comprobar dos hechos:

- 1) " \cdot " es asociativa. En efecto, como " $*$ " es asociativa (pues $\langle M, * \rangle$ es un monoide),

$$[a] \cdot \{ [b] \cdot [c] \} = [a] \cdot [b * c] = [a * (b * c)] = [(a * b) * c] = \{ [a] \cdot [b] \} \cdot [c]$$

- 2) Existe un elemento neutro. En efecto, si el elemento neutro en $\langle M, * \rangle$ es e , tendremos que

$$[a] \cdot [e] = [a * e] = [a]$$

y

$$[e] \cdot [a] = [e * a] = [a]$$

por lo que $[e]$ es el elemento neutro para $\langle M/R, \cdot \rangle$.

Definición 4.2.4.6. El monoide $\langle M/R, \cdot \rangle$ se denomina *monoide cociente* de M por R .

4.3. Comportamiento de entrada-estados

Sea un AF

$$A = \langle E, S, Q, f, g \rangle$$

Sabemos que dado un símbolo de entrada y un estado podemos obtener el estado siguiente mediante la función

$$f: E \times Q \rightarrow Q$$

Dado solamente un símbolo de entrada, $e \in E$, podemos definir una función que aplique a cada estado el siguiente bajo esa entrada determinada, es decir, una función de Q en Q , $k(e): Q \rightarrow Q$. Existirá entonces una función

$$k: E \rightarrow Q^Q$$

siendo Q^Q el conjunto de las funciones de Q en Q , que sabemos por el Teorema 4.2.1.1 que, con la composición de funciones, es un monoide. La función k se determina a partir de f del siguiente modo: para cada $e \in E$, y cada $q \in Q$, $[k(e)](q) = f(e, q)$.

Ahora bien, por el Teorema 4.2.3.1, la función k puede extenderse a un homomorfismo entre monoides:

$$K: \langle E^*, \cdot \rangle \rightarrow \langle Q^Q, \circ \rangle,$$

con $K(e_1 e_2 \dots e_n) = k(e_1) \circ k(e_2) \circ \dots \circ k(e_n)$. Llamaremos a K *comportamiento de entrada-estados* del autómata.

Ejemplo. Consideremos el autómata detector de paridad descrito por el diagrama de la figura 2.6. La función $k: \{0, 1\} \rightarrow Q^Q$ se obtiene fácilmente del diagrama y se puede resumir en forma de tabla:

		Estado siguiente	
		$k(0)$	$k(1)$
Estado inicial	q_1	q_1	q_2
	q_2	q_2	q_1

Calculemos K para algunas cadenas:

$$K(0) = k(0); K(1) = k(1);$$

$$K(00) = k(0) \circ k(0) = k(0)$$

$$K(01) = k(0) \circ k(1) = k(1)$$

$$K(10) = k(1) \circ k(0) = k(1)$$

$$K(11) = k(1) \circ k(1) = k(0)$$

etcétera.

En general,

$$K(x) = k(0) \text{ si se tiene un número par de unos}$$

$$K(x) = k(1) \text{ si se tiene un número impar de unos}$$

Si la cadena es vacía, $K(\lambda)$ será la función identidad en Q , es decir, la función que aplica q_1 en q_1 y q_2 en q_2 , que coincide con $k(0)$.

4.4. Relación equirrespuesta y monoide de un autómeta

Sea un autómeta A con comportamiento de entrada-estados K .

Definición 4.4.1. La relación equirrespuesta de A , \equiv , es una relación binaria en E^* tal que

$$(\forall x, y \in E^*) [(x \equiv y) \leftrightarrow (K(x) = K(y))]$$

\equiv es una relación de equivalencia, ya que se cumple:

$$(\forall x \in E^*) [K(x) = K(x)]$$

$$(\forall x, y \in E^*) [(K(x) = K(y)) \rightarrow (K(y) = K(x))]$$

$$(\forall x, y, z \in E^*) [(K(x) = K(y)) \wedge (K(y) = K(z)) \rightarrow (K(x) = K(z))]$$

Además es una relación de congruencia en el monoide $\langle E^*, >$. En efecto, si $x \equiv y$, $K(x) = K(y)$, y $K(xz) = K(x) \circ K(z) = K(y) \circ K(z) = K(yz)$. Análogamente, $K(zx) = K(zy)$. Por tanto, según el Teorema 4.2.4.5., $\langle E^* / \equiv, >$, es decir, el conjunto cociente E^* / \equiv con la operación de concatenación, es un monoide.

Definición 4.4.2. Dado un autómata con un comportamiento de entrada-estados K que origina una relación equirrespuesta \equiv en E^* , el monoide cociente $\langle E^* / \equiv, \cdot \rangle$ se llama *monoide del autómata*.

Si el número de estados es n , el monoide del autómata tendrá como máximo n^n elementos. En efecto, el número de transformaciones en el conjunto Q , es decir, de funciones diferentes $Q \rightarrow Q$ es n^n . El homomorfismo K aplica entonces un conjunto infinito, E^* , en un conjunto, Q^Q , que tiene n^n elementos. Si esta aplicación es suprayectiva la relación equirrespuesta tendrá índice n^n , es decir, inducirá n^n clases de equivalencia en E^* (si la aplicación no fuera suprayectiva, es decir, si existieran una o más funciones $Q \rightarrow Q$ a las que no correspondiese ningún elemento de E^* , el número de clases de equivalencia sería menor). Por consiguiente, el número máximo de elementos del conjunto cociente E^* / \equiv es n^n .

El monoide de un autómata refleja la capacidad de éste para responder de distinto modo a las cadenas de entrada. En efecto, en E^* hay infinitas cadenas, mientras que en E^* / \equiv hay como máximo n^n elementos, que son las clases de congruencia de \equiv . Si dos cadenas diferentes, x e y , están en la misma clase, es decir, $x \equiv y$, entonces $K(x) = K(y)$, es decir, el homomorfismo K las aplica sobre el mismo elemento de Q^Q , o lo que es lo mismo, ambas producen la misma transformación $Q \rightarrow Q$, y el autómata será incapaz de distinguir una de la otra.

4.5. Ejemplos

4.5.1. Detector de paridad

Ya hemos estudiado el comportamiento de entrada-estados del detector de paridad, llegando a la conclusión de que sólo hay dos clases de equivalencia en E^* ; en efecto, veíamos que $K(x) = k(0) = k(\lambda)$, si x tiene un número par de unos (o ninguno) y $K(x) = k(1)$, si x tiene un número impar de unos. Luego toda cadena x es equivalente a " λ " o a " 1 ". Llamemos $[\lambda]$ y $[1]$ a estas dos clases de equivalencia, que serán los elementos del monoide del autómata. Al tener un número finito de elementos, podemos describirlo mediante una tabla que indique el elemento resultante de la concatenación de otros dos:

	$[\lambda]$	$[1]$
$[\lambda]$	$[\lambda]$	$[1]$
$[1]$	$[1]$	$[\lambda]$

4.5.2. Reconocedor de la cadena 010

El autómata, representado en la figura 2.12, tiene cuatro estados, por lo que el número total posible de transformaciones en Q , es decir, el número máximo de elementos del monoide del autómata es $4^4 = 256$. Veamos si existen todos. Para ello, calculemos $K(x)$ para cadenas de longitud creciente desde $x = \lambda$. Resumimos los resultados en la tabla de la figura 2.15. El procedimiento para construir una tabla de ese tipo es el siguiente:

Tenemos tantas filas como estados tiene la máquina. Para cada cadena x ponemos en una columna los estados resultantes de la función $K(x): Q \rightarrow Q$. Comenzamos por λ , para la que la función es la unidad (es decir, q_1 en q_1 , q_2 en q_2 , etc). Para las cadenas de longitud 1 (en este caso 0 y 1) miramos en el diagrama de Moore; en este ejemplo, cuando se aplica $x = 0$ la imagen de q_1 es q_2 , la de q_2 es q_3 , etc. Teniendo ya $K(x)$ para cadenas de longitud 1 no hace falta consultar más el diagrama; por ejemplo, $K(01) = K(0) \circ K(1)$, y así, con 0, q_1 se aplica en q_2 , y luego con 1, q_2 se aplica en q_3 , por lo que la imagen de q_1 será q_3 . Para cadenas de longitud 3 nos basamos en los resultados anteriores; por ejemplo, $K(010) = K(01) \circ K(0) = K(0) \circ K(10)$ (las dos formas de hacerlo son válidas). La tabla se va así rellenando por columnas, aumentando la longitud de las cadenas por postconcatenación de los símbolos del alfabeto a las cadenas ya tratadas, hasta que encontramos cadenas x tales que $K(x) = K(y)$, donde $\lg(y) \leq \lg(x)$; entonces podemos asegurar que $x \equiv y$, y no es preciso que sigamos aumentando x , ya que, si la concatenamos con un símbolo cualquiera, e , tendremos $K(xe) = K(ye)$, es decir, $xe \equiv ye$, con $\lg(ye) \leq \lg(x)$, por lo que la clase de ye o bien ya ha aparecido (si $\lg(y) < \lg(x)$), o bien va a aparecer (si $\lg(y) = \lg(x)$).

Estado inicial	x	λ	0	1	00	01	10	11	000	001	010
	q_1	q_1	q_2	q_1	q_2	q_3	q_2	q_1	q_2	q_3	q_4
	q_2	q_2	q_2	q_3	q_2	q_3	q_4	q_1	q_2	q_3	q_4
	q_3	q_3	q_4	q_1	q_2	q_3	q_2	q_1	q_2	q_3	q_4
	q_4	q_4	q_2	q_3	q_2	q_3	q_4	q_1	q_2	q_3	q_4

x	011	100	101	110	111	0100	0101
q_1	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_2	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_3	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_4	q_1	q_2	q_3	q_2	q_1	q_2	q_3

Figura 2.15.

Concretemos con nuestro ejemplo. Siete de las ocho cadenas de longitud 3 tienen el mismo comportamiento que las cadenas de longitud 2: $K(000) = K(00)$; $K(001) = K(01)$, etc., por lo que $[000] = [00]$; $[001] = [01]$, etc. La única cadena cuyo comportamiento difiere de los anteriores es 010. Por tanto, estudiamos 0100 y 0101, y vemos que $K(0100) = K(00)$ y $K(0101) = K(001)$. Nos quedan así solamente ocho clases de equivalencia: $[\lambda]$, $[0]$, $[1]$, $[00]$, $[01]$, $[10]$, $[11]$ y $[010]$. El monoide del autómata tendrá pues ocho elementos, no 256: K dista mucho de ser suprayectiva. La tabla del monoide se obtiene sin dificultad de la figura 2.15, y es la representada en la figura 2.16; por ejemplo, para saber el resultado de $[11][10]$, con ayuda de la figura 2.15 vemos que $K(1110) = K(11) \circ K(10) = K(00)$, por lo que $[11][10] = [00]$.

	$[\lambda]$	$[0]$	$[1]$	$[00]$	$[01]$	$[10]$	$[11]$	$[010]$
$[\lambda]$	$[\lambda]$	$[0]$	$[1]$	$[00]$	$[01]$	$[10]$	$[11]$	$[010]$
$[0]$	$[0]$	$[00]$	$[01]$	$[00]$	$[01]$	$[010]$	$[11]$	$[010]$
$[1]$	$[1]$	$[10]$	$[11]$	$[00]$	$[01]$	$[00]$	$[11]$	$[010]$
$[00]$	$[00]$	$[00]$	$[01]$	$[00]$	$[01]$	$[10]$	$[11]$	$[010]$
$[01]$	$[01]$	$[010]$	$[11]$	$[00]$	$[01]$	$[00]$	$[11]$	$[010]$
$[10]$	$[10]$	$[00]$	$[01]$	$[00]$	$[01]$	$[010]$	$[11]$	$[010]$
$[11]$	$[11]$	$[00]$	$[11]$	$[00]$	$[01]$	$[00]$	$[11]$	$[010]$
$[010]$	$[010]$	$[00]$	$[01]$	$[00]$	$[01]$	$[010]$	$[11]$	$[010]$

Figura 2.16.

Con la tabla del monoide se puede hallar rápidamente el comportamiento para cualquier cadena. Así, por ejemplo, si queremos conocer $C_q(010000110)$ tendremos:

$$[010000110] = [010][00][01][10] = [00][00] = [00],$$

y como $[00]$ transforma cualquier estado inicial en q_2 (figura 2.15), y q_2 da salida 0, tendremos que $C_q(010000110) = 0$, $i = 1, 2, 3, 4$.

4.5.3. Contador en código binario natural módulo 10

Un contador es un autómata que proporciona en cada momento información sobre el número de entradas de un determinado tipo recibidas hasta ese momento. Supondremos que $E = \{0, 1\}$, y que el contador debe dar como salida el código en BCD natural correspondiente al número total de "1" recibidos. Si el autómata

es finito sólo podrá contar hasta un determinado número y a partir de él volver al principio; un contador módulo p contará de 0 a $p - 1$.

Consideremos $p = 10$; es decir, el contador contará de 0 a 9 y al recibir el dígito "1" volverá a contar desde 0. Cada vez que recibe un "1" deberá cambiar de estado (y, naturalmente, de salida). Así es fácil ver que el diagrama de Moore es el de la figura 2.17, en donde se supone que el estado inicial es q_0 , y como símbolos de salida se han tomado s_0, \dots, s_9 , para abreviar la notación (en realidad serían 0000, 0001, ..., 1001).

Como hay diez estados, el monoide del autómata podría tener hasta 10^{10} elementos. Ahora bien, si construimos la tabla de $K(x)$ (figura 2.18) vemos que sólo hay diez clases de equivalencia.

En efecto, vemos que $0 \equiv \lambda$ por lo que sólo hay que considerar cadenas compuestas por "1". Calculamos $K(1^2)$, $K(1^3)$, etc. (el exponente indica concatenación: $1^3=111$, etc.), y vemos que $1^{10} \equiv \lambda$. Por consiguiente, las clases de equivalencia son:

$$[\lambda], [1], [1^2], [1^3], \dots, [1^9]$$

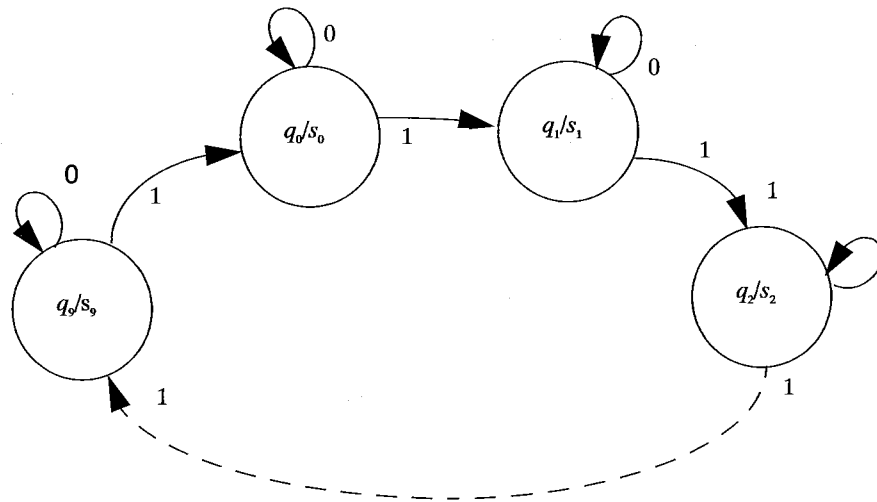


Figura 2.17.

Dejamos como ejercicio al lector la obtención de la tabla del monoide del autómata. Con ella podrá fácilmente comprobar que cualquier cadena que con-

tenga n_0 "ceros" y n_1 "unos" en cualquier orden pertenece a la clase de equivalencia $[1^m]$, donde $m=n_1 \pmod{10}$.

	λ	0	1	1^2	\dots	1^9	1^{10}
q_0	q_0	q_0	q_1	q_2		q_9	q_0
q_1	q_1	q_1	q_2	q_3		q_0	q_1
$:$	$:$	$:$	$:$	$:$		$:$	$:$
$:$	$:$	$:$	$:$	$:$		$:$	$:$
q_8	q_8	q_8	q_9	q_0		q_7	q_8
q_9	q_9	q_9	q_0	q_1		q_8	q_9

Figura 2.18.

4.6. Relación entre el comportamiento de entrada-salida y el comportamiento de entrada-estados

El comportamiento de entrada-salida lo hemos definido para un estado inicial q como:

$$C_q: E^* \rightarrow S$$

Así, para cada entrada, x , $C_q(x)$ es la función que aplica x en $g(x, q)$:

$$C_q(x): x \rightarrow g(x, q)$$

El comportamiento de entrada-estados, sin embargo, se define para todo el conjunto de estados:

$$K: E^* \rightarrow Q^Q$$

De modo que, para cada entrada, x , $K(x)$ aplicará x en una función de transformación entre estados:

$$K(x): x \rightarrow l: Q \rightarrow Q,$$

donde la función l está determinada por la función f restringida a la entrada x :

$$f: E^* \times Q \rightarrow Q,$$

$$f(x): Q \rightarrow Q$$

El sentido físico es el siguiente: C_q determina, dado un estado inicial, el símbolo final de salida para cada cadena de entrada; K determina, para cada cadena de entrada, la correspondiente transformación entre estados, es decir, nos dice que de q_1 se pasará a q_p , de q_2 a q_p , etc. Por tanto, K nos proporciona más información acerca del comportamiento interno del autómata. Por otra parte, la relación de equivalencia en E^* inducida por K y el monoide del autómata resultante nos permiten deducir fácilmente C_q para cualquier cadena de entrada. Así, en el ejemplo del reconocedor de la cadena 010 veíamos que $[010000100] = [00]$, y de aquí que $C_q = 0$; para llegar a la misma conclusión con el procedimiento que seguíamos en el apartado 3.5 hubiera sido preciso reconocer sobre el diagrama (figura 2.12) todas las transiciones producidas por la cadena.

Por otra parte, podría pensarse en estudiar el comportamiento de entrada-salida para todo el conjunto de estados. Para ello podemos definir un comportamiento global de entrada-salida, $K_s(x)$, como una aplicación de E^* en el conjunto de funciones $l: Q_0 \rightarrow S_r$, donde Q_0 son los estados iniciales y S_r las salidas finales. K_s no será ahora un homomorfismo entre monoides, como lo era K , ya que el conjunto de funciones l no es un monoide; por tanto, K_s no nos permite definir un monoide de la máquina, y por ello hemos utilizado K en el análisis anterior. En cualquier caso, K_s también nos define una relación de equivalencia en E^* , que podría llamarse "relación equisalida", \equiv_s , tal que $x \equiv_s y$ si y sólo si $K_s(x) = K_s(y)$. \equiv_s particiona E^* en clases de equivalencia. Ahora bien, suponiendo máquinas de Moore, tenemos una función $h: Q_r \rightarrow S_r$ que nos aplica el estado final en la salida final correspondiente a ese estado, y por tanto, si llamamos $[x]_i$ a la clase de equivalencia de E^* / \equiv que conduce a una determinada transformación $k_i: Q_0 \rightarrow Q_r$ y $[y]_j$ a la clase de equivalencia de E^* / \equiv_s que corresponde a una determinada $l_j: Q_0 \rightarrow S_r$, tendremos:

$$[y]_j = \cup [x]_i \text{ para todo } i \text{ tal que } k_i \circ h = l_j$$

Esto demuestra que las clases de equivalencia de \equiv_s están perfectamente determinadas por \equiv y h , y justifica lo que decíamos en nota a pie de página en el apartado 4.1: que al considerar sólo la respuesta de los estados el estudio no pierde generalidad.

5. Minimización de un autómata finito

5.1. Planteamiento del problema

En el ejemplo 3.5.1 veíamos dos AF equivalentes, uno de los cuales estaba en forma mínima. Si estos AF describen un sistema secuencial que debe realizarse físicamente escogeremos el que está en forma mínima, ya que el coste de la realización, como se verá en el capítulo 3, crece con el número de estados. Es, pues, importante poder saber si un AF está en forma mínima, y, si no lo está, hallar un AF en forma mínima equivalente a él. Comenzaremos por ver que éste existe siempre; a continuación veremos que para detectar equivalencia entre estados no es preciso realizar infinitos ensayos con cadenas de entrada, y, finalmente, veremos un algoritmo para minimizar un AF, es decir, para hallar otro AF en forma mínima equivalente. Ante todo, debemos señalar que la equivalencia entre estados de un AF, definida por

$$(\forall x \in E^*) [(q_1 = q_2) \leftrightarrow (C_{q_1}(x) = C_{q_2}(x))]$$

es una relación de equivalencia en el conjunto Q , pues, como es inmediato comprobar, es reflexiva, simétrica y transitiva. Por consiguiente, puede definirse un conjunto cociente, Q / \equiv , cuyos elementos serán las clases de equivalencia: $[q]$ será la clase que contiene a q .

5.2. Autómata en forma mínima de un autómata dado

Vamos a demostrar que, dado $A = \langle E, S, Q, f, h \rangle$, el AF definido por $A_M = \langle E, S, Q_M, f_M, h_M \rangle$, donde

$$Q_M = Q / \equiv$$

$$f_M(x, [q]) = [f(x, q)]$$

$$h_M([q]) = h(q)$$

es equivalente a A y está en forma mínima.

En primer lugar, habrá que demostrar que f_M y h_M están bien definidas, es decir, que son independientes del elemento q que se tome dentro de la clase $[q]$, o, lo que es lo mismo, que si $q_1 \equiv q_2$, entonces $(\forall x) ([f(x, q_1)] = [f(x, q_2)])$, y $h(q_1) = h(q_2)$. En efecto, por la definición de la equivalencia, $(\forall x \in E^*) [(q_1 = q_2) \rightarrow (g(x, q_1) = g(x, q_2))]$. Si tomamos $x = \lambda$ resulta $h(q_1) = h(q_2)$. Y si tomamos $x = x_1x_2$,

$$g(x_1x_2, q_1) = g(x_2, f(x_1, q_1))$$

$$g(x_1x_2, q_2) = g(x_2, f(x_1, q_2))$$

por lo que $(\forall x_1) ([f(x_1, q_1) = f(x_1, q_2)])$, es decir, $(\forall x \in E^*) ([f(x, q_1)] = [f(x, q_2)])$.

Por otra parte, es evidente que A y A_M son equivalentes, puesto que todo estado q_i de A será equivalente al estado $[q_i]$ de A_M .

Finalmente, A_M está en forma mínima, ya que si los estados $[q_1]$ y $[q_2]$ de A_M fueran equivalentes, q_1 y q_2 de A deberían ser también equivalentes, por lo que $[q_1] = [q_2]$.

5.3. Comprobación de la equivalencia entre estados de un autómata

5.3.1. Teorema de las particiones sucesivas

Teorema 5.3.1.1. Si P_0, P_1, P_2, \dots es una secuencia infinita de particiones en un conjunto finito Q tal que, para todo k , se cumple:

- a) P_{k+1} es un *refinamiento* de P_k , es decir, todo bloque de P_{k+1} , B_{k+1}^i , está contenido en un bloque, B_k^j , de P_k : $B_{k+1}^i \subset B_k^j$.
- b) $(P_{k+1} = P_k) \rightarrow (P_{k+2} = P_{k+1})$,

entonces existe un número entero $k_0 < \text{card}(Q)$ tal que $(\forall k \geq k_0) (P_k = P_{k_0})$. Para demostrarlo, supongamos que $\text{card}(P_0) = 0$, $\text{card}(P_1) = 1$, $\text{card}(P_2) = 2, \dots$. Pero como, para todo k , $\text{card}(P_k) \leq \text{card}(Q) = n$ (finito), deberá existir un entero $k_0 < n$ tal que $\text{card}(P_{k_0+1}) = \text{card}(P_{k_0})$, es decir, $P_{k_0+1} = P_{k_0}$, y, por la condición b), $(\forall k \geq k_0) (P_k = P_{k_0})$.

5.3.2. Equivalencia de orden k entre estados

Definición 5.3.2.1. Dos estados de un AF son *equivalentes de orden k* si y sólo si conducen a la misma salida para cadenas de entrada de longitud igual o inferior a k :

$$(q_1 \equiv_k q_2) \leftrightarrow [(\text{lg}(x) \leq k) \rightarrow C_{q_1}(x) = C_{q_2}(x)]$$

Obsérvese que para comprobar la equivalencia de orden k ya no hay que hacer un número infinito de comprobaciones o "experimentos".

Teorema 5.3.2.2. Dado un AF y $q_1, q_2 \in Q$, existe un $k_0 < \text{card}(Q)$ tal que q_1 y q_2 son equivalentes ($q_1 \equiv q_2$) si y sólo si q_1 y q_2 son equivalentes de orden k_0 ($q_1 \equiv_{k_0} q_2$).

Para demostrar este teorema nos apoyaremos en el 5.3.1.1. Veamos pues que las particiones inducidas en Q por las sucesivas equivalencias de orden 0, 1, ..., k , $k+1$... cumplen las condiciones a) y b) de aquel teorema.

- a) Si $q_1 \equiv_{k+1} q_2$, entonces $(\forall x \text{ lg}(x) \leq k+1) [C_{q_1}(x) = C_{q_2}(x)]$ y, evidentemente $\forall x \text{ lg}(x) \leq k$, por lo que $q_1 \equiv_k q_2$. Por tanto, todo bloque de la partición P_{k+1} inducida por \equiv_{k+1} está contenido en un bloque de P_k , es decir, P_{k+1} es un refinamiento de P_k .
- b) Supongamos que $P_{k+1} = P_k$, es decir, $(q_1 \equiv_k q_2) \rightarrow (q_1 \equiv_{k+1} q_2)$. Si q_1 y q_2 son equivalentes de orden $k+1$, los estados $f(e, q_1)$ y $f(e, q_2)$ serán, sea cual sea e , equivalentes de orden k , pero como $P_{k+1} = P_k$ también serán equivalentes de orden $k+1$, y por ello q_1 y q_2 son equivalentes de orden $k+2$, de donde $P_{k+2} = P_{k+1}$. (El razonamiento es una sucesión de condicionales. Definiendo, para abreviar la notación, las variables proposicionales

$$a: P_{k+1} = P_k$$

$$b: P_{k+2} = P_{k+1}$$

$$c: q_1 \equiv_k q_2$$

$$c': q_1 \equiv_{k+1} q_2$$

$$c'': q_1 \equiv_{k+2} q_2$$

$$d: (\forall e \in E) (f(e, q_1) \equiv_k f(e, q_2))$$

$$d': (\forall e \in E) (f(e, q_1) \equiv_{k+1} f(e, q_2))$$

podemos expresarlo formalmente así:

$$[(a \rightarrow (c \rightarrow c')) \wedge (c' \rightarrow d) \wedge (a \rightarrow (d \rightarrow d') \wedge (c' \rightarrow c'')) \rightarrow \\ \rightarrow ((c' \rightarrow c'') \rightarrow b)] \rightarrow (a \rightarrow b),$$

que puede comprobarse que es una tautología).

En definitiva, se cumplen las condiciones a) y b) del Teorema 5.3.1.1, y, por tanto, existe $k_0 < \text{card}(Q) = n$ tal que $P_k = P_{k_0}$, $k \geq k_0$, con lo que $(q_1 \equiv_{k_0} q_2) \rightarrow (q_1 \equiv q_2)$.

La conclusión importante es que, sin necesidad de conocer k_0 , como $k_0 < n$ (números de estados del AF), para comprobar si dos estados son equivalentes bastará con comprobar si son equivalentes de orden $n - 1$.

5.4. Algoritmo para minimización de un autómata finito

El algoritmo para hallar el AF en forma mínima de un AF dado se basa en los resultados anteriores:

1. $k = 0$. Formar P_0 , poniendo en el mismo bloque los estados que tengan asociada la misma salida ($q_i \equiv_0 q_j$ si y sólo si $h(q_i) = h(q_j)$).
2. Formar P_{k+1} , teniendo en cuenta que dos estados estarán en el mismo bloque de P_{k+1} si y sólo si para cada entrada los estados siguientes están en el mismo bloque de P_k ($q_i \equiv_{k+1} q_j$ si y sólo si $f(e, q_i) \equiv_k f(e, q_j)$, $\forall e \in E \cup \{\lambda\}$).
3. Si $P_{k+1} \neq P_k$ incrementar k en una unidad y volver al paso 2.
4. $k_0 = k$. Proceso terminado. El AF en forma mínima tiene como estados $Q_M = Q / \equiv_{k_0}$.

5.5. Ejemplos

Ejemplo 5.5.1

Consideremos el autómata reconocedor de 010 descrito en 3.5.1, y partamos del diagrama no mínimo (figura 2.13). A efectos de aplicación del algoritmo es más cómodo representar el AF por la tabla de transiciones:

	0	1
$q_1/0$	q_2	q_5
$q_2/0$	q_2	q_3
$q_3/0$	q_4	q_5
$q_4/1$	q_2	q_3
$q_5/0$	q_6	q_5
$q_6/0$	q_6	q_7
$q_7/0$	q_4	q_5
$q_8/1$	q_6	q_7

1. Observando las salidas asociadas a cada estado podemos poner:

$$P_0 = \{q_1, q_2, q_3, q_5, q_6, q_7\}, \{q_4, q_8\}$$

2. Con entrada 0, de q_1, q_2, q_5 y q_6 se pasa a estados del primer bloque de P_0 , y con 1 también. De q_3 y q_7 , con 0 se pasa a estados del segundo bloque de P_0 , y con 1 a otros del primer bloque. Finalmente, de q_4 y q_8 , con 0 se pasa al primer bloque de P_0 y con 1 también. Luego

$$P_1 = \{q_1, q_2, q_5, q_6\}, \{q_3, q_7\}, \{q_4, q_8\}$$

- 2'. En lugar de con palabras, pongamos simbólicamente los resultados:

$$\left. \begin{array}{l} f(0, q_1) = q_2 \\ f(0, q_2) = q_2 \\ f(0, q_5) = q_6 \\ f(0, q_6) = q_6 \end{array} \right\} \begin{array}{l} \\ (1^{\text{er}} \text{ bloque}) \\ \\ \end{array} \left\{ \begin{array}{ll} f(1, q_1) = q_5 & (1^{\text{er}} \text{ bloque}) \\ f(1, q_2) = q_3 & (2^{\circ} \text{ bloque}) \\ f(1, q_5) = q_6 & (1^{\text{er}} \text{ bloque}) \\ f(1, q_6) = q_7 & (2^{\circ} \text{ bloque}) \end{array} \right.$$

$$\left. \begin{array}{l} f(0, q_3) = q_4 \\ f(0, q_7) = q_4 \end{array} \right\} (3^{\text{er}} \text{ bloque}) \left\{ \begin{array}{ll} f(1, q_3) = q_5 & \\ f(1, q_7) = q_5 & \end{array} \right\} (1^{\text{er}} \text{ bloque})$$

$$\left. \begin{array}{l} f(0, q_4) = q_2 \\ f(0, q_8) = q_6 \end{array} \right\} (1^{\text{er}} \text{ bloque}) \left\{ \begin{array}{ll} f(1, q_4) = q_3 & \\ f(1, q_8) = q_7 & \end{array} \right\} (2^{\circ} \text{ bloque})$$

Luego,

$$P_2 = \{q_1, q_5\}, \{q_2, q_6\}, \{q_3, q_7\}, \{q_4, q_8\}$$

- 2''. Si repetimos la operación, viendo a qué bloques de P_2 se pasa con cada entrada veremos que $P_3 = P_2$. Luego $k_0 = 2$, y el AF minimizado tiene como estados los bloques definidos en P_2 ; dando los nuevos nombres $\{q_1, q_5\} = q_1$, etc., se obtienen el diagrama de la figura 2.12.

Ejemplo 5.5.2

Sea el AF dado por el diagrama de la figura 2.19 y su tabla:

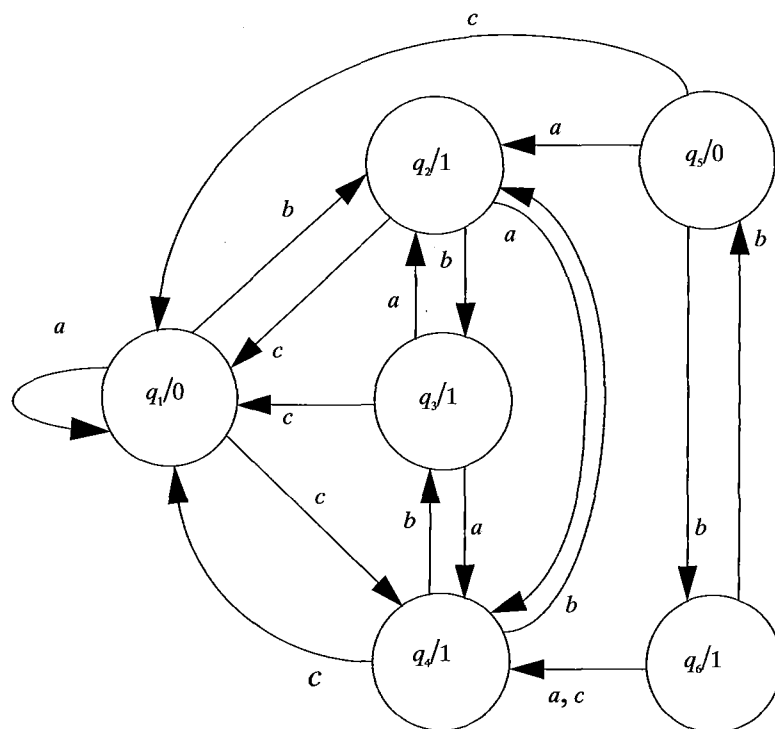


Figura 2.19.

	<i>a</i>	<i>b</i>	<i>c</i>
$q_1/0$	q_1	q_2	q_4
$q_2/1$	q_4	q_3	q_1
$q_3/1$	q_2	q_4	q_1
$q_4/1$	q_3	q_2	q_1
$q_5/0$	q_2	q_6	q_1
$q_6/1$	q_4	q_5	q_4

Siguiendo el algoritmo, encontramos sucesivamente:

$$P_0 = \{q_1, q_5\}, \{q_2, q_3, q_4, q_6\}$$

$$P_1 = \{q_1\}, \{q_5\}, \{q_2, q_3, q_4\}, \{q_6\}$$

$$P_2 = P_1$$

Llamando $q_1 = \{q_1\}$; $q_2 = \{q_2, q_3, q_4\}$; $q_3 = \{q_5\}$ y $q_4 = \{q_6\}$ obtendremos el diagrama de la figura 2.20, que representa el AF en forma mínima equivalente al dado.

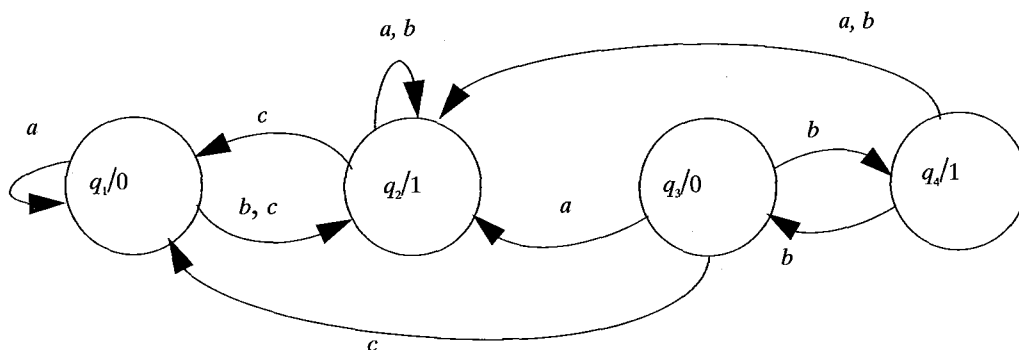


Figura 2.20.

6. Resumen

Hemos expuesto la teoría básica de los autómatas finitos procurando buscar la mayor generalidad, generalidad que se conserva aunque el estudio se restrinja a la máquina de Moore.

El comportamiento de entrada-salida es un concepto importante en relación con los lenguajes, que utilizaremos en el capítulo sobre autómatas reconocedores. Otros conceptos importantes son los de equivalencia entre estados y entre autómatas, pero el interés de éstos radica más bien en la realización tecnológica.

El estudio del comportamiento global del autómata, es decir, inicializado en cualquier estado, nos ha llevado a establecer el concepto de monoide del autómata, estructura algebraica que refleja su capacidad para responder de distinto modo a cadenas diferentes de entrada. Finalmente, hemos visto cómo se puede comprobar la equivalencia entre diferentes estados con un número finito de experimentos, lo que nos ha permitido establecer un algoritmo para obtener el AF en forma mínima equivalente a un AF dado.

7. Notas histórica y bibliográfica

Quizás el primero en expresar el concepto moderno de autómata haya sido Leonardo Torres Quevedo, que, en 1915, respondía así en una entrevista de una revista americana: "Los antiguos autómatas... imitaban el aspecto y los movimien-

tos de los seres vivos, pero eso no tiene mucho interés en la práctica, y lo que buscamos es un tipo de aparato que deje los meros gestos visibles del hombre e intente conseguir los resultados que obtiene una persona, para, de este modo, reemplazar a un hombre por una máquina" ("Torres and his remarkable automatic devices", *Scientific American*, nº 113, 6 nov. 1915, p.296).

El primer estudio riguroso sobre autómatas fue el publicado por Moore (1956). Con anterioridad, debido al desarrollo de los primeros ordenadores, se habían estudiado diversos métodos para la síntesis de circuitos secuenciales (Huffman, 1954; Mealy, 1955). A finales de los años 50 se comenzó a ver la utilidad de los autómatas en relación con los lenguajes, y la mayor parte de los trabajos sobre teoría de autómatas finitos se realizó durante los años 60. Por esta razón, las referencias más importantes que pueden darse sobre este campo son libros publicados entre 1965 y 1970. El de Harrison (1965) cubre tanto la parte combinacional como los circuitos secuenciales, con un tratamiento muy riguroso y fácil de seguir, aunque se limita a estudiar autómatas reconocedores. Otra obra recomendable es la de Booth (1967), que, además de autómatas finitos, trata también las máquinas de Turing, lenguajes artificiales y autómatas estocásticos. Muchas de las definiciones y terminología de este capítulo las hemos tomado de Arbib (1969) que cubre muy ampliamente los diversos estudios desarrollados hasta aquella fecha. La parte que trata del monoide del autómata está basada en el capítulo sobre monoides de Gilbert (1976), libro muy interesante sobre aplicaciones del álgebra moderna. Hay textos más recientes, como el de Shields (1987), pero la teoría básica de autómatas se presenta actualmente integrada en obras más generales sobre la teoría de la computación, como las de Lewis y Papadimitriou (1988) y Cohen (1991).

Para no alargar excesivamente el tema hemos presentado el algoritmo de minimización reducido al caso de máquinas de Moore (la máquina de Mealy mínima tendrá normalmente, menos estados). El caso general viene expuesto en cualquiera de los libros citados en éste o en el siguiente capítulo.

8. Ejercicios

- 8.1 Obtener la tabla de transiciones y el diagrama de Moore de la máquina de Moore equivalente a la de Mealy, descrita por la siguiente tabla:

$q \backslash e$	e_1	e_2
q_1	q_3/s_1	q_2/s_2
q_2	q_4/s_2	q_3/s_2
q_3	q_4/s_1	q_2/s_2
q_4	q_2/s_2	q_4/s_1

- 8.2. Definir las máquinas de Mealy y de Moore de un restador binario.
- 8.3. Considérese un sistema eléctrico cuyas entradas son dos pulsadores, a y c (apertura y cierre), que pueden estar en reposo (0) o pulsados (1) (se supone que no puede ser $a = c = 1$), y cuyas salidas son tres lámparas: verde, ámbar y roja (V, A, R), de las que, en cada momento, una y sólo una está encendida. Inicialmente, está encendida la verde; al pulsar c , ha de pasar de verde a ámbar, y, si se pulsa otra vez (o más veces), a rojo. El pulsador a hace pasar siempre a verde.
- a) Dibujar el diagrama de Moore, con $Q = \{V, A, R\}$ y $E = \{a, c\}$.
- b) El modelo anterior tiene un inconveniente: si la transición de un estado a otro es instantánea (o casi), entonces el hecho de pulsar C estando en verde va a provocar el paso a rojo sin detenerse (casi) en ámbar. Las especificaciones dadas se pueden completar diciendo que hasta que no "se suelte" el pulsador no puede hacerse otra transición. Ahora va a ser necesario un estado más: "ámbar con $c = 1$ " y "ámbar con $c = 0$ ". Dibujar el nuevo diagrama de Moore.
- 8.4. Obtener los monoides del sumador y del restador binario y compararlos.
- 8.5. Un autómata retardador es aquel en que

$$s(t) = e(t - n)$$

Suponiendo $E = S = \{0, 1\}$ y $n = 2$, obtener el diagrama de Moore y el monoide del autómata.

- 8.6. Obtener el monoide del autómata cuya tabla de transición es:

$q/s \backslash e$	a	b	c
$q_1/1$	q_2	q_2	q_3
$q_2/0$	q_1	q_2	q_3
$q_3/1$	q_3	q_3	q_2

- 8.7. (Gilbert, 1976). En primavera, un brote de planta requiere unas condiciones adecuadas para su desarrollo. En una determinada especie, el brote necesita que se presente un día de lluvia seguido de dos días calurosos sin ser interrumpido por ningún día frío o de helada. Además, si hay un día de helada después de que se haya desarrollado el brote, éste muere. Dibujar el diagrama de Moore de este proceso. Puede

utilizarse $E = \{L, C, F, H\}$, donde L quiere decir "día lluvioso", C "caluroso", etc., y $S = \{T, B, M\}$, donde T = "latente", B = "brote", M = "muerto". ¿Cuál es el número de elementos en el monoide de este autómata?

- 8.8.** (Gilbert, 1976). Un perro puede estar tranquilo, irritado, asustado, o irritado y asustado, en cuyo caso muerde. Si le damos un hueso queda tranquilo. Si le quitamos uno de sus huesos se pone irritado, y, si ya estaba asustado, nos muerde. Si le amenazamos se asusta y, si ya estaba irritado nos muerde. Obtener el diagrama de Moore y el monoide del perro.
- 8.9.** Dibujar un diagrama de Moore para un autómata reconocedor de la cadena "321", suponiendo que $E = \{1,2,3\}$. Obtener el monoide del autómata. Comprobar si está en forma mínima.
- 8.10.** Considérese un autómata finito definido por

$$E = \{a, b\}; S = \{0, 1\}; Q = \{q_1, q_2, q_3, q_4\}$$

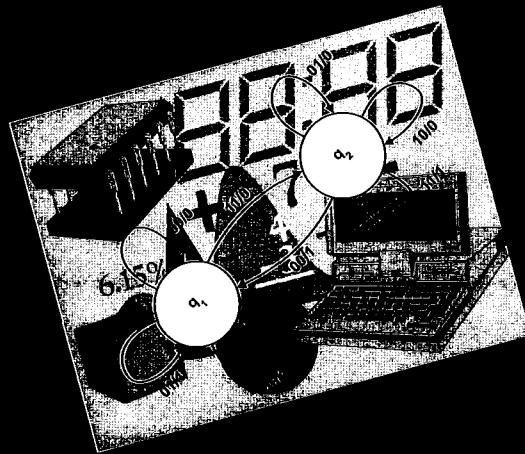
y la tabla de transiciones:

$q \backslash \theta$	a	b
q_1	$q_2/1$	$q_1/0$
q_2	$q_3/0$	$q_4/0$
q_3	$q_3/0$	$q_1/1$
q_4	$q_3/0$	$q_1/1$

- 1º Este autómata, ¿es una máquina de Moore o de Mealy? ¿Por qué?
- 2º Si es una máquina de Mealy, dibujar el diagrama de Moore de la máquina de Moore equivalente.
- 3º Minimizar el autómata resultante de la pregunta anterior.
- 4º Si se ha seguido el algoritmo de minimización expuesto aquí se habrá obtenido la máquina de Moore en forma mínima. ¿Existe una máquina de Mealy equivalente que tenga menos estados? Si es así, dibujar su diagrama de Moore.

Fundamentos de informática

3



3

Circuitos secuenciales

1. La realización de autómatas finitos

Los AF que hemos estudiado en el capítulo anterior no son, en definitiva, más que modelos matemáticos para sistemas digitales con memoria. Ya mencionábamos en el capítulo 1 el doble interés de su estudio: teórico (herramienta matemática para formalizar el estudio de los lenguajes) y práctico (realización de sistemas digitales que cumplen funciones especificadas).

La realización de un sistema digital puede enfocarse de dos maneras:

- a) Puede construirse físicamente el sistema, partiendo de elementos sencillos.
- b) Puede utilizarse un ordenador, que es un sistema digital de uso general, y programarse para que efectúe las funciones que se desean.

Por ejemplo, si el autómata que se quiere realizar es un ordenador, se seguirá, normalmente, el enfoque a). (También puede seguirse el enfoque b), y en ese caso se trataría de simulación o emulación en otro ordenador). Si el autómata es muy complicado (como es un ordenador), se descompondrá generalmente en subautómatas; uno de ellos podría ser el sumador binario serial, ya estudiado, del que veremos su realización en el apartado 6.1. (Conviene señalar que los ordenadores suelen utilizar otro tipo de sumador, el paralelo, que exige unos circuitos más complicados, pero que es mucho más rápido).

Otro ejemplo puede ser el reconocimiento de cadenas. Hemos visto un AF que reconoce la cadena "010". Podemos pensar en un AF que reconozca varias cadenas, dando una salida diferente para cada una. Así, si $E = \{A, B, \dots, Z\}$ y $S = \{00000, 00001, \dots, 11111\}$, podría dar $s = 00000$ para $x = \text{CAR}$ (cargar), $s = 00001$ para $x = \text{CMP}$ (complementar), $s = 00010$ para $x = \text{SUM}$ (sumar) y así, sucesivamente, todos los códigos binarios de operación de algún ordenador para los correspondientes códigos de su lenguaje ensamblador. Este AF sería parte de un AF más completo, llamado "ensamblador", que traduciría a lenguaje de máquina los programas escritos en lenguaje ensamblador. Pues bien, los autómatas ensambladores, caso particular de procesadores de lenguajes, no se construyen físicamente, sino mediante un programa de ordenador.

En este capítulo nos vamos a dedicar especialmente a las técnicas para realizar autómatas finitos físicamente. La realización del AF, salvo raras excepciones, es con tecnología electrónica o electromecánica, y el sistema digital con memoria resultante se llama circuito secuencial.

2. Elementos de un circuito secuencial

2.1. Tipos de elementos

La realización de las funciones definidas por un AF implica dos tipos de elementos:

- a) Elementos combinacionales, que realizan las funciones lógicas en las que no interviene el tiempo. Por ejemplo, la función $h: Q \rightarrow S$ debe dar la salida asociada a cada estado, sin intervención del tiempo, por lo que es una función lógica puramente combinacional, representable por una tabla de verdad y realizable con los métodos estudiados en el capítulo 3 del tema "Lógica".
- b) Elementos con memoria, para realizar las funciones en las que interviene el tiempo. Así, $f: E \times Q \rightarrow Q$ debe dar, para unos valores dados de E y Q en el instante t el valor resultante de Q en $t + 1$ (recordemos que este "1" se refiere a una escala de tiempos arbitraria), es decir, debe calcular el valor resultante de Q y memorizarlo para darlo en $t + 1$.

Pasemos a estudiar el funcionamiento de algunos de estos elementos.

2.2. Elementos combinacionales

Son las puertas NOT, AND, OR, NAND, NOR, etc., ya estudiadas en el tema "Lógica" (capítulo 3). Realizadas generalmente mediante tecnología electrónica, para aplicaciones especiales aún se utilizan otras tecnologías: electromecánica, hidráulica, neumática, etc.

2.3. Elementos con memoria

2.3.1. Líneas de retardo

El elemento con memoria más sencillo que existe (desde el punto de vista conceptual) es aquel en que se tiene $E = Q = S = \{0, 1\}$ y la salida en cualquier instante es igual al estado en ese instante es igual a la entrada retardada un intervalo de tiempo θ :

$$s(t + \theta) = q(t + \theta) = e(t)$$

con $e, q, s \in \{0, 1\}$. Este sencillo modo de funcionamiento se puede ilustrar gráficamente con un cronograma como el de la figura 3.1.

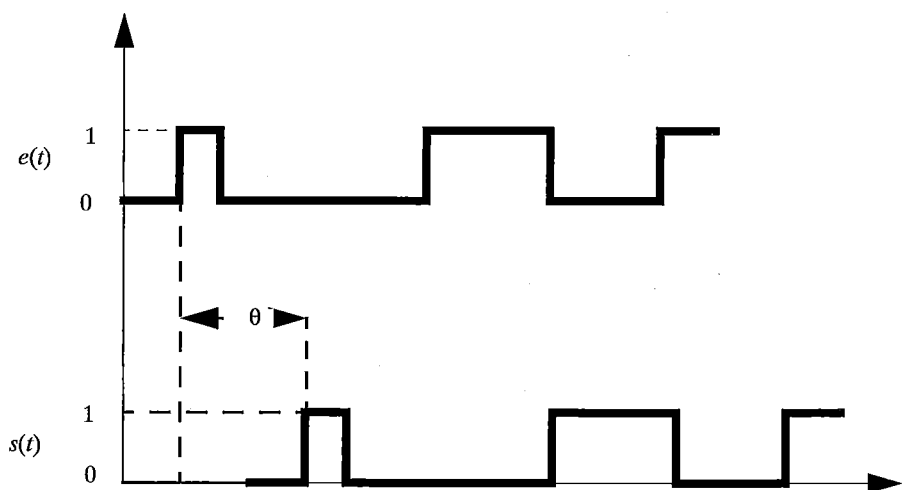


Figura 3.1.

Estos elementos (que, de una manera general, se llaman líneas de retardo) pueden realizarse mediante líneas de retardo acústicas, líneas de transmisión, redes de condensadores e inductancias, puertas lógicas conectadas en serie, etc., y también con biestables, como veremos enseguida.

2.3.2. Biestables

Los elementos de memoria más utilizados en los circuitos secuenciales son los biestables electrónicos. Un biestable es un circuito con dos estados estables que son también su salida y que, simbólicamente, se denominan "0" y "1" (en la realidad, cada uno de ellos corresponderá a un determinado nivel de tensión). "Estables" quiere decir que el elemento sólo cambia de estado bajo la acción de las entradas. Hay diversos tipos de biestables, que difieren entre sí por los posibles símbolos de entrada y las funciones de transición. Todos ellos tienen dos líneas de salida: en una de ellas, denominada habitualmente " Q ", se tiene el nivel de tensión correspondiente en cada momento al estado, 0 ó 1; en la otra, \bar{Q} , se tiene su complemento. Pasemos a ver el funcionamiento de algunos biestables.

Biestable tipo SR ("set-reset"). En este biestable $E = \{00, 01, 10\}$. Como las señales en los circuitos secuenciales son binarias, tendrá dos líneas de entrada, S y R (figura 3.2 a). Su funcionamiento puede expresarse formalmente mediante la función de transición:

$$Q(t+1) = S(t) + \bar{R}(t) \cdot Q(t) \quad (R \cdot S = 0)$$

(aquí la unidad de tiempo es el intervalo que tarda el biestable en cambiar de estado o "tiempo de basculamiento"), o gráficamente, mediante el diagrama de Moore (figura 3.2 b), o mediante un cronograma (figura 3.2 c). Expresado con palabras, el funcionamiento es:

- $S = 0, R = 0$ hace que el biestable no cambie de estado.
- $S = 1, R = 0$ pone 1 a ("set") el biestable.
- $S = 0, R = 1$ lo pone a 0 o repone ("reset").
- $S = 1, R = 1$ es una entrada no permitida.

Es fácil diseñar, con los métodos ya conocidos, un circuito lógico que realice al biestable RS : basta minimizar la función de transición, función de S, R y Q y realimentar $Q(t+1)$ a la entrada. Un posible circuito es el de la figura 3.2d.

\bar{Q} Biestable tipo JK. Es como el SR , sólo que aquí la entrada $J = 1, K = 1$ está permitida, y su efecto consiste en cambiar el estado que tuviera anteriormente el biestable (figura 3.3; en el cronograma, para no complicar demasiado su interpretación, se ha despreciado el tiempo de basculamiento).

Hay que advertir que este biestable no se utiliza nunca con la realización de la figura 3.3 d, sino sincronizado, como se verá más adelante. La razón es que si se mantienen las entradas $J = K = 1$ el circuito se convierte en un oscilador con una frecuencia que depende del tiempo de basculamiento del biestable.

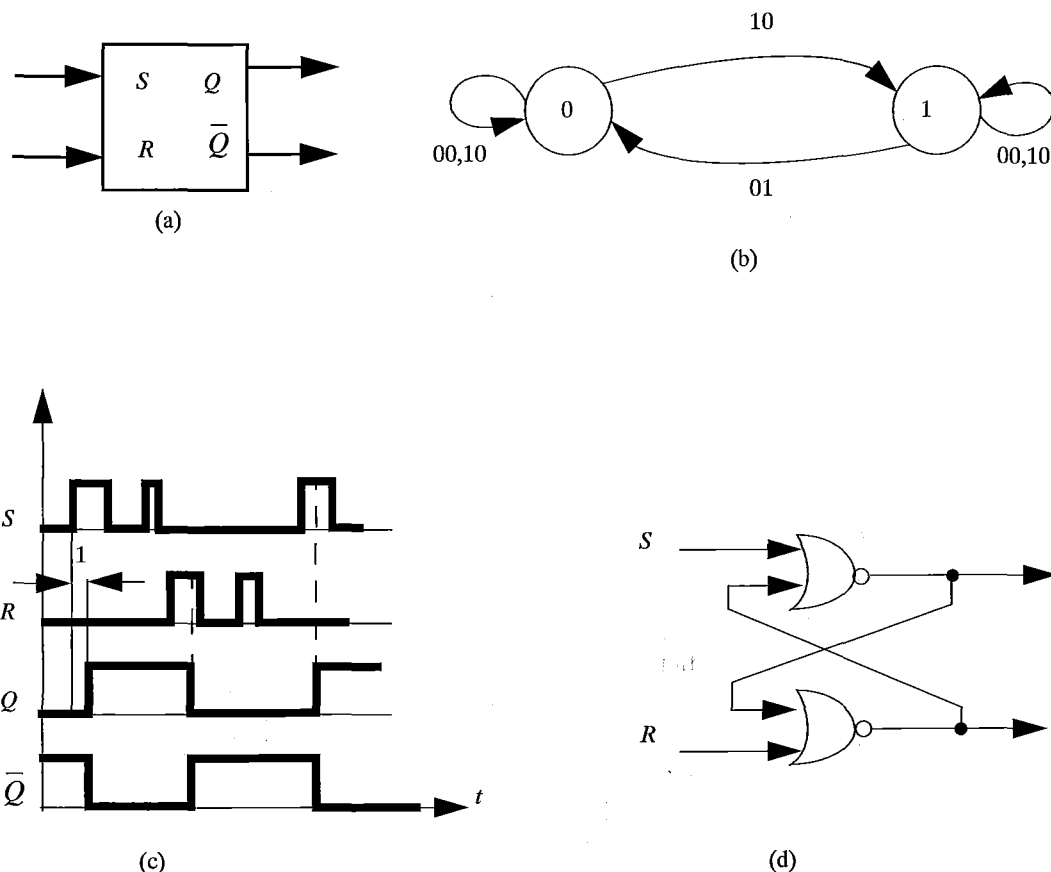


Figura 3.2.

Biestable tipo D. También tiene dos líneas de entrada, ya que $E = \{00, 01, 10, 11\}$. Estas líneas se llaman D ("data") y C ("clock") (figura 3.4 a). Su funcionamiento puede representarse formalmente así:

$$Q(t+1) = D(t) \cdot C(t) + Q(t) \cdot \bar{C}(t)$$

que corresponde al diagrama de la figura 3.4 b. Ahora bien, aquí hay una diferencia muy importante en cuanto al significado de los símbolos "0" y "1" para D y para C . Para D , "0" significa, por ejemplo, una tensión de 0 v, y "1", 5 v; sin embargo, para C "1" significa que hay *una variación de tensión* de 0 v a 5 v, y "0" que no hay tal variación. En otras palabras, el valor de C es cero *salvo en el instante* en que su tensión cambia de nivel bajo a nivel alto. Véase la figura 3.4 c, para ilustrar este significado de "0" y "1".

Un cronograma que ilustra el funcionamiento del biestable tipo D es el de la figura 3.4 d, donde hemos, también, despreciado el tiempo de basculamiento.

Obsérvese que si los cambios en D van retardados ligeramente un intervalo constante $\varepsilon \ll \theta$ con relación a los cambios de 0 a 1 de C , lo que hace el biestable es exactamente retardar la entrada D en θ segundos, siendo θ el período del reloj (C).

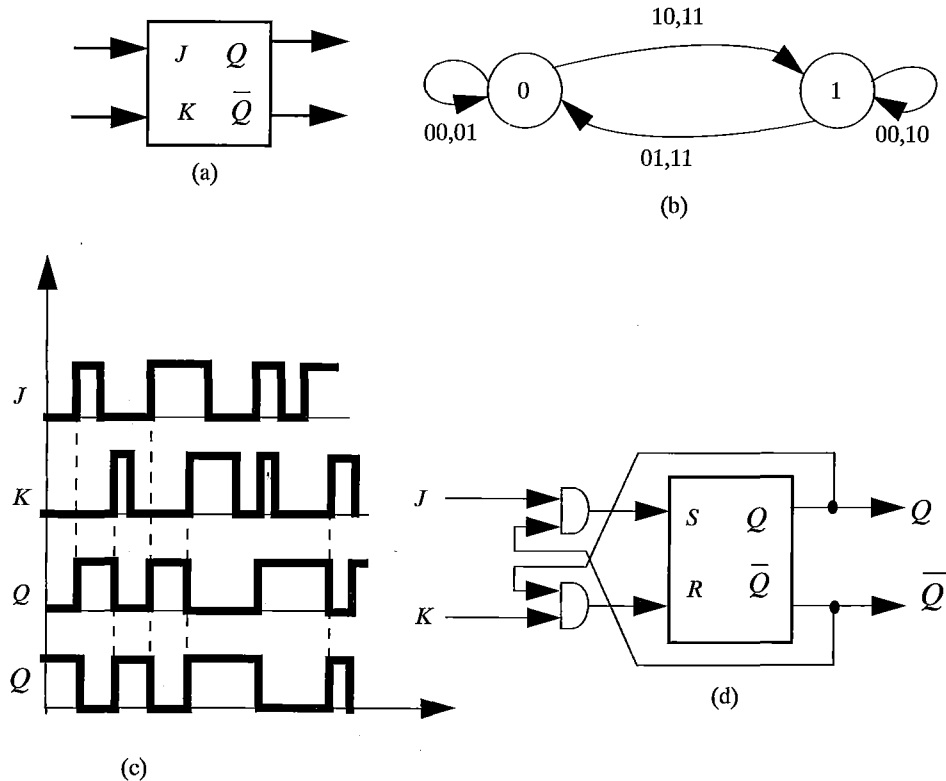


Figura 3.3.

Este tipo de biestable, que sólo cambia de estado cuando cambia el nivel de la señal de reloj se llama *sincronizado por flancos* ("edge triggered"). El que hemos visto está sincronizado por el flanco de subida; también lo hay sincronizado por el flanco de bajada (es decir, cambia de estado cuando C pasa de 1 a 0). No entramos ya en su circuitería lógica, que el lector interesado puede consultar en la bibliografía.

Biestable tipo JK sincronizado. Tiene cinco líneas de entrada: S , R , J , K y C (figura 3.5 a). S y R se utilizan para ponerlo a 1 ó a 0 de una manera asíncrona, es decir, independientemente de C , mientras que J y K son la puesta a 1 ó a 0 síncronas. La ecuación que describe su comportamiento es:

$$Q(t+1) = S + \bar{R}\bar{K}Q + \bar{R}\bar{C}Q + \bar{R}JC\bar{Q}$$

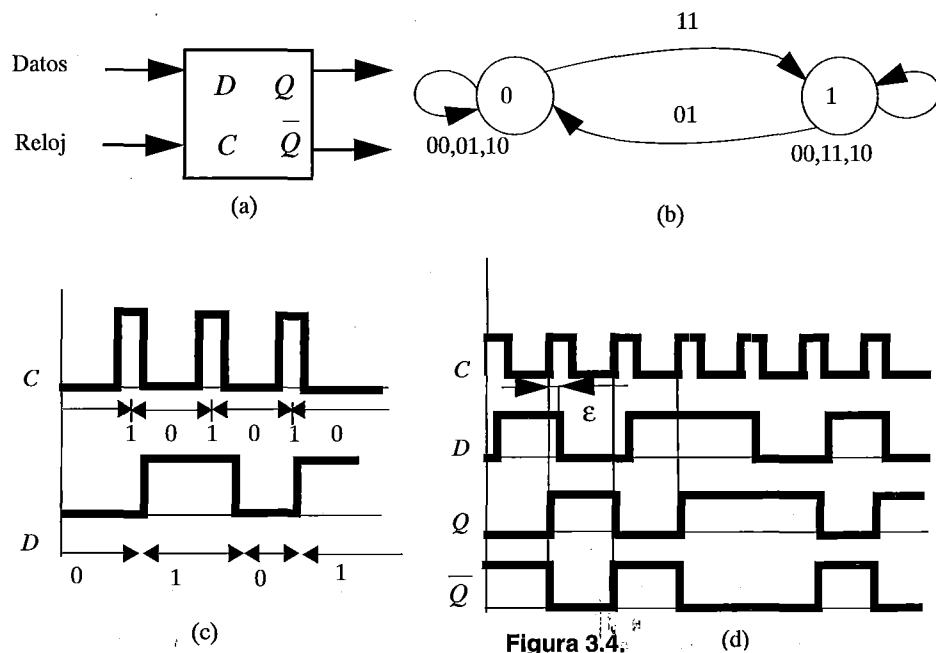


Figura 3.4.

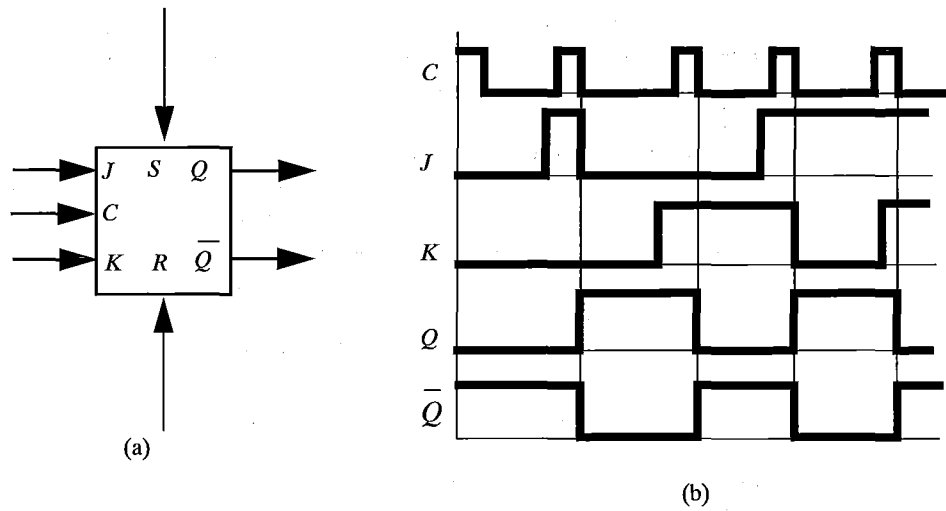


Figura 3.5.

Aquí el valor lógico "1" para C significa una doble transición nivel bajo-nivel alto-nivel bajo. Esto es debido a la constitución interna, en la que no entraremos aquí, que incluye dos biestables en una configuración conocida como "maestro-

esclavo". Por ello, las transiciones de la señal de salida, Q , se dan en el flanco de bajada de C . El cronograma de la figura 3.5 b ilustra el funcionamiento síncrono (S y R actúan como en un SR normal, y, en caso de conflicto, predominan sobre las JK).

3. Modelos básicos de circuitos secuenciales

Recordemos que las funciones que definen un AF son:

$$q(t+1) = f(e(t), q(t))$$

$$s(t) = g(e(t), q(t))$$

De ellas podemos deducir un primer modelo general para circuitos secuenciales (figura 3.6 a); consiste en disociar la parte combinacional, realizable mediante circuitos lógicos que calculan f y g , de la parte de memoria, que retarda en una unidad de tiempo $q(t)$ y $s(t)$. Este sería el modelo de Mealy, porque si existe una función de salida $h: Q \rightarrow S$ tal que $s(t) = h(q(t))$, podemos establecer el modelo de Moore (figura 3.6 b).

Supongamos que E tiene tres símbolos, a, b, c ; como los circuitos combinacionales son binarios, tendremos que codificarlos, haciendo, por ejemplo, $a = 00$, $b = 01$, $c = 10$. Por tanto, la entrada " e " será en realidad, 2 hilos de entrada binaria: uno para el primer dígito del código, y otro para el segundo. En general, si E tiene l elementos, tendremos n hilos, con n tal que $2^{n-1} < l \leq 2^n$. Del mismo modo, habremos de suponer que en la realización con tecnología binaria serán necesarios, en general, m hilos para la salida y p hilos para el estado. De acuerdo con esto, el modelo básico de Mealy en forma de diagrama de bloques será el de la figura 3.7, en donde ya se supone que por todos los hilos las señales son binarias, y de igual modo podríamos dibujar el modelo básico de Moore. Para prescindir en la escritura de la variable t hemos adoptado en convenio, que seguiremos en adelante, de llamar z_i a $q_i(t+1)$.

Con estas nuevas variables binarias la función f será:

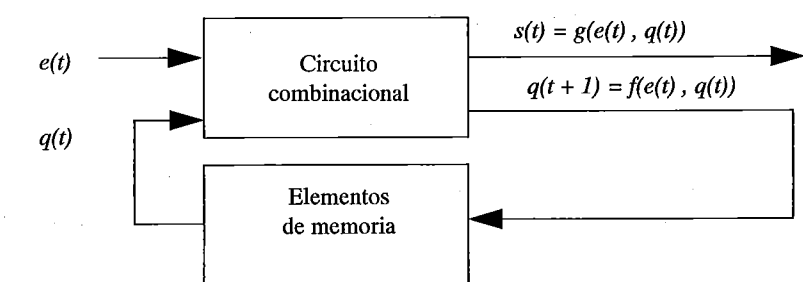
$$z_i = f_i(e_1, \dots, e_n, q_1, \dots, q_p), \quad i = 1, \dots, m$$

y análogamente g y h . Si definimos los vectores columna $\mathbf{z}, \mathbf{q}, \mathbf{e}, \mathbf{s}$ tendremos con notación vectorial:

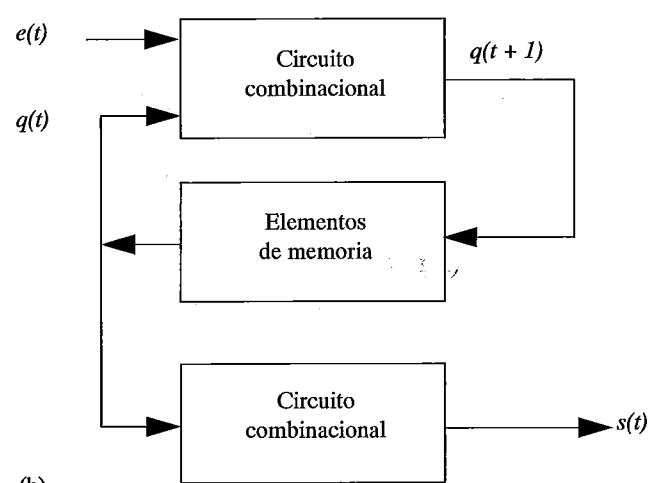
$$\mathbf{z} = f(\mathbf{e}, \mathbf{q})$$

$$\mathbf{s} = g(\mathbf{e}, \mathbf{q})$$

$$\mathbf{s} = h(\mathbf{q})$$



(a)



(b)

Figura 3.6.

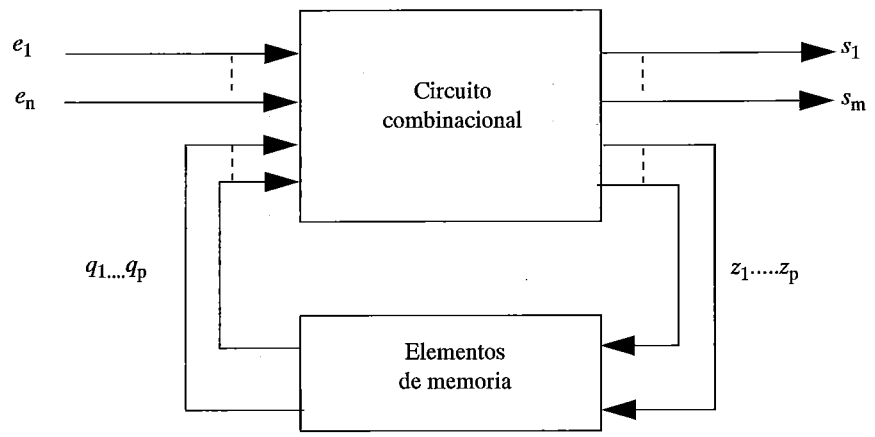


Figura 3.7.

4. Tipos de circuitos secuenciales

En el apartado 2.3 hemos visto dos tipos de elementos con memoria: los no sincronizados, y los sincronizados con una señal de reloj, *C*. Por otra parte, las señales de entrada pueden ser de cuatro tipos: impulsionales síncronas o asíncronas y de nivel síncronas o asíncronas (figura 3.8). Si se utilizan unidades de memoria sincronizadas lo normal es que las señales de entrada sean síncronas, y así tenemos cuatro tipos de circuitos secuenciales:

- Tipo 1. *Síncronos impulsionales*. (Elementos de memoria sincronizados y entradas impulsionales síncronas).
- Tipo 2. *Síncronos de nivel*. (Igual que los anteriores, pero entradas de nivel).
- Tipo 3. *Asíncronos impulsionales*. (Elementos de memoria no sincronizados y entradas impulsionales asíncronas).
- Tipo 4. *Asíncronos de nivel*. (Como el tipo 3, con entradas de nivel).

Las salidas serán síncronas (impulsionales o de nivel) para el tipo 1, y lo mismo para el 2, asíncronas (impulsionales o de nivel) para el tipo 3 y necesariamente asíncronas de nivel para el tipo 4.

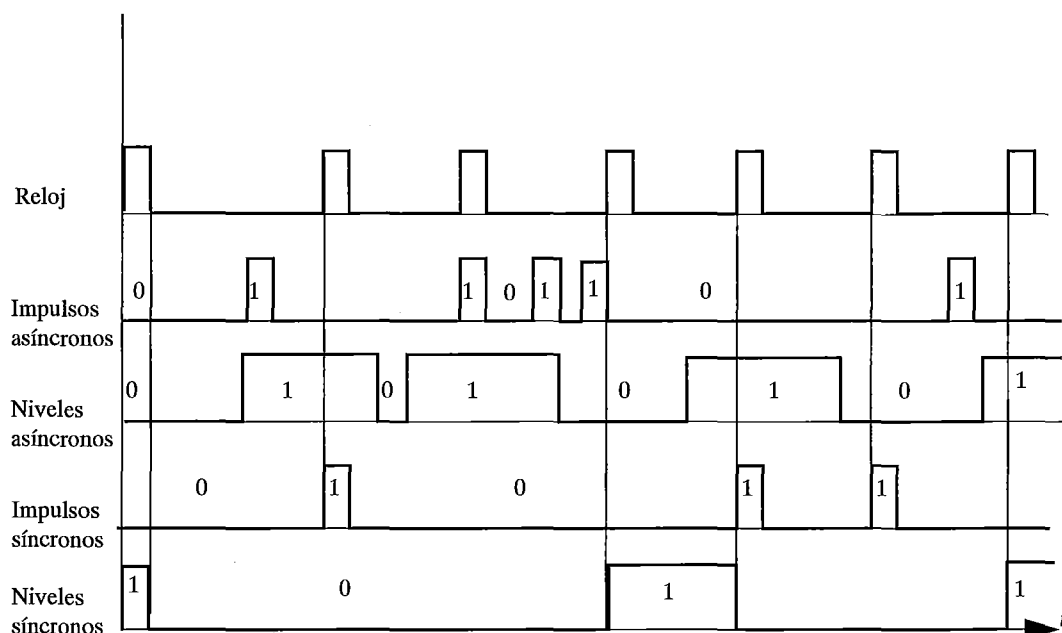


Figura 3.8.

Dependiendo de las características particulares de cada aplicación se utiliza un tipo u otro de circuito secuencial. Aquí nos vamos a limitar para los ejemplos exclusivamente a los tipos 1 y 2.

5. Análisis de circuitos secuenciales

El análisis de un circuito secuencial consiste en obtener su salida para una determinada cadena de entrada, o bien obtener su representación ya sea formal, ya en diagramas o tablas. El problema es conceptualmente muy fácil, y vamos a ilustrarlo con un ejemplo sencillo.

Consideremos el circuito secuencial de la figura 3.9.

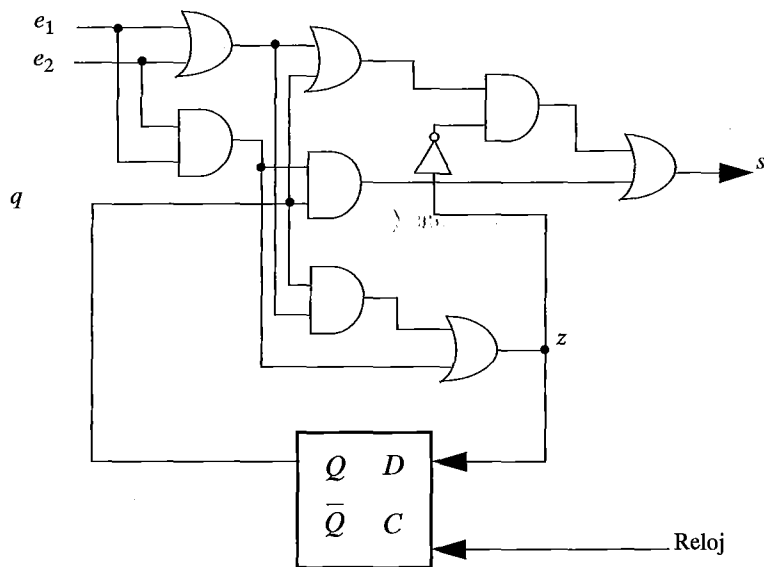


Figura 3.9.

De la parte combinacional vemos que $s = e_1 \cdot e_2 \cdot q + \bar{z} \cdot (e_1 + e_2 + q)$ y $z = e_1 \cdot e_2 + q \cdot (e_1 + e_2)$, y de la parte de memoria, como es un biestable tipo D, $q = z \cdot C + q \cdot \bar{C}$, es decir, q mantiene su valor hasta que C pasa de 0 a 1, instante en el que q pasa a valer z . Supongamos que las señales de entrada son de nivel (sincronizadas con C), que el estado inicial del biestable es $q = 0$, y que introducimos las cadenas de entrada $x_1 = 100110$, $x_2 = 101110$. Durante el primer intervalo de tiempo $q = 0$, tenemos $z = 1 \cdot 1 + 0 \cdot (1 + 1) = 1$, $s = 1 \cdot 1 \cdot 0 + 0 \cdot (1 + 1 + 0) = 0$. Al llegar el siguiente impulso de reloj q pasa a valer 1 (pues $z = 1$), $z = 0 \cdot 0 + 1 \cdot (0 + 0) = 0$; $s = 0 \cdot 0 \cdot 1 + 1 \cdot (0 + 0 + 1) = 1$. Si vamos

anotando gráficamente estos resultados obtenemos el cronograma de la figura 3.10.

El cronograma es muy útil para ver gráficamente el comportamiento del circuito, pero sólo lo representa para una determinada cadena de entrada. Para tener una representación más general de este circuito podemos poner en forma de tabla todas las posibles combinaciones de e_1 , e_2 y q y las resultantes para s y z . Esta tabla (figura 3.11 a) nos conduce inmediatamente a la tabla de transición (figura 3.11 b) y al diagrama de Moore (figura 3.11 c). Obsérvese que este diagrama es precisamente el del autómata sumador binario, es decir, el circuito de la figura 3.9 es la realización física de tal autómata. (La parte combinacional es una etapa de sumador).

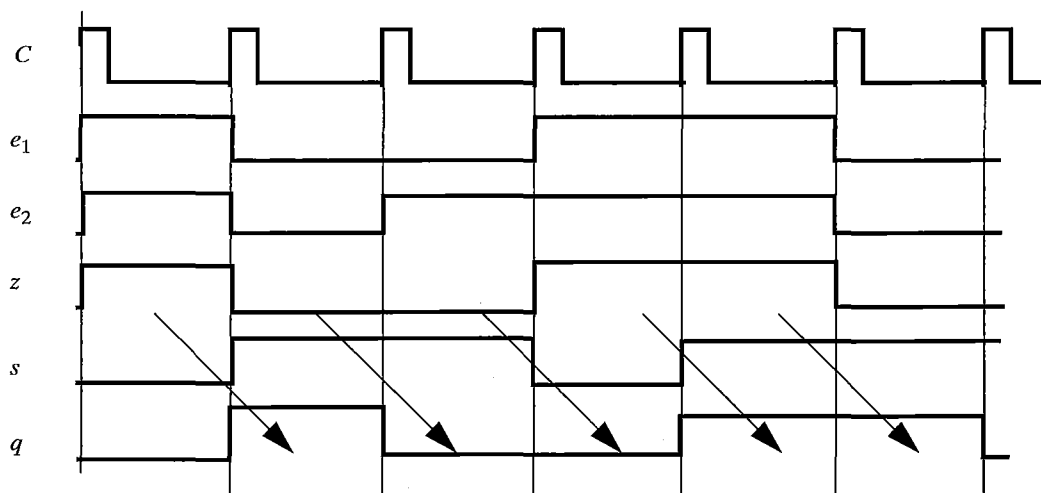


Figura 3.10.

6. Síntesis de circuitos secuenciales

6.1. Pasos de la síntesis

El diseño de un circuito secuencial se lleva a cabo siguiendo los siguientes pasos:

- a) A partir de las especificaciones, obtener una tabla de transiciones y/o un diagrama de Moore.

e_1	e_2	q	s	z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$q \backslash e$	00	01	10	11
0	0/0	0/1	0/1	1/0
1	0/1	1/0	1/0	1/1

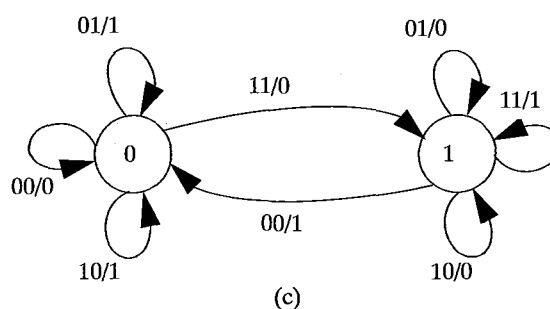


Figura 3.11.

- Minimizar el AF.
- Hacer una asignación de estados.
- Escribir la tabla de transición en binario.
- En función del tipo de biestable utilizado, obtener las tablas de excitación, que dan las entradas necesarias al biestable para cada transición.
- De las tablas de excitación, obtener las ecuaciones lógicas de la parte combinacional.
- Obtener las ecuaciones lógicas de $s = h(q)$.

Dada la diversidad de posibles especificaciones, no existe un método para el paso a).

Para el paso b) puede seguirse el algoritmo estudiado en el apartado 5 del capítulo 2.

El paso c) significa lo siguiente: Supongamos que el AF minimizado en el paso b) tiene N estados. Como los elementos de memoria de que disponemos son

binarios, necesitaremos p elementos, con $2^{p-1} < N \leq 2^p$. La asignación de estados es una codificación arbitraria de los N estados con p dígitos binarios. El problema aquí es que según se haga una u otra codificación de las $(2^p!) / (2^p - N)!$ posibles, el circuito combinacional resultante puede ser más o menos complicado, y que no existe un método para saber cuál es la codificación óptima, aunque sí hay algunas reglas en las que no vamos entrar, que el lector interesado puede estudiar en la bibliografía.

El paso e) consiste en obtener las tablas de verdad de la parte combinacional.

Para los pasos f) y g) aplicaremos las técnicas ya conocidas de diseño de circuitos lógicos.

6.2. Ejemplos

6.2.1. Detector de paridad

El diagrama de Moore es el de la figura 2.6. Si asignamos a los estados los valores $q_1 = 0$ y $q_2 = 1$, la tabla de transición en binario será:

$q \backslash e$	0	1
0/0	0	1
1/1	1	0

Escribamos la tabla de una forma lineal, es decir, con una línea para cada combinación posible de e y q :

e	q	z
0	0	0
0	1	1
1	0	1
1	1	0

Supongamos ahora que vamos a utilizar un biestable tipo D (sólo hace falta uno, puesto que sólo hay dos estados). Deberemos anotar en cada línea la entrada necesaria al biestable para que tenga lugar la transición correspondiente. Por ejemplo, en la primera línea, de $q(t) = 0$ se debe pasar a $z = q(t+1) = 0$. Para ello, $D = 0$; en la segunda línea $D = 1$, etc. Así, podemos poner la tabla:

e	q	D
0	0	0
0	1	1
1	0	1
1	1	0

que nos define D (entrada del biestable) en función de $e(t)$ y $q(t)$, y se llama *tabla de excitaciones*. De ella obtenemos la función D :

$$D = \bar{e} \cdot q + e \cdot \bar{q} = e \oplus q$$

y, como $s = q$, la realización del circuito será la de la figura 3.12 a, o bien la de la figura 3.12 b. En realidad, hemos tomado $s = z$, lo que nos permite obtener s con un intervalo de adelanto.

Obsérvese que, debido al modo de funcionamiento del biestable D , la tabla de excitaciones se obtiene, simplemente, sustituyendo "z" por "D", ya que la salida del biestable en $t + 1$ es justamente su entrada en t . No ocurre lo mismo con el JK . En efecto, si tenemos que realizar la transición de $q = 0$ a $z = 0$, las entradas J y K

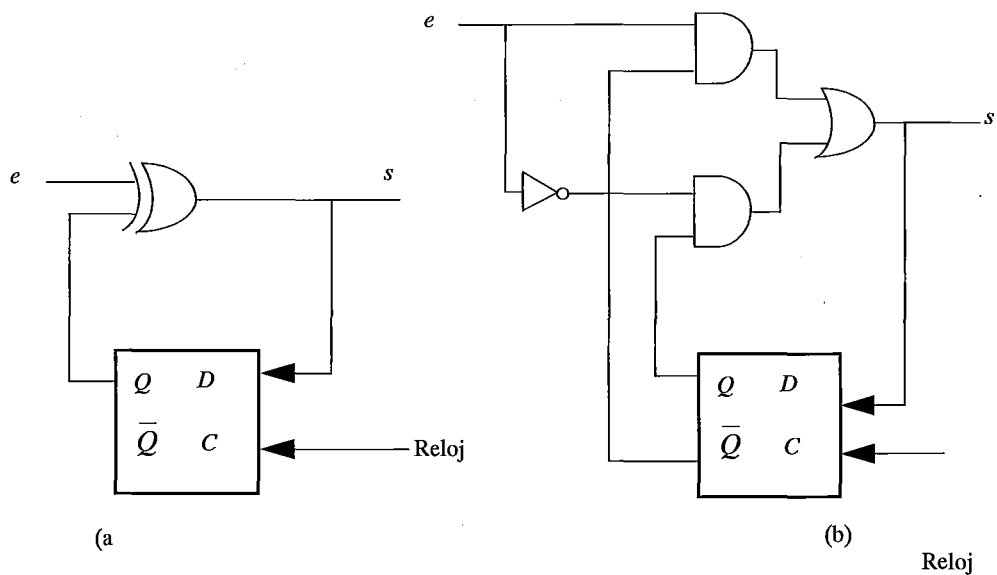
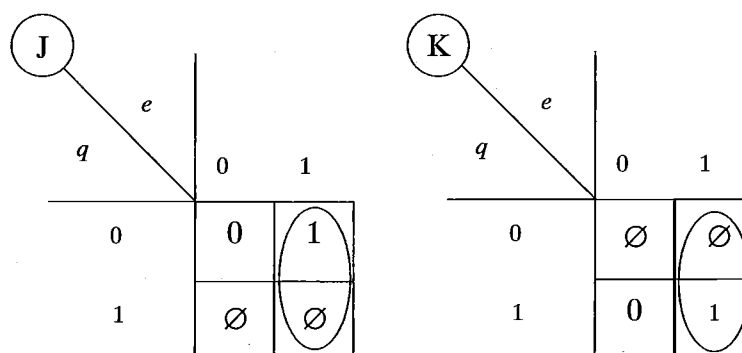


Figura 3.12.

pueden ser ambas 0, o bien $J = 0, K = 1$ (puesta a cero); para la transición de $q = 0$ a $z = 1$ podemos hacer $J = 1, K = 0$ (puesta a uno), o bien $J = 1, K = 1$ (complementación), etc. Así, podemos escribir la tabla de excitaciones para este ejemplo:

e	q	z	J	K
0	0	0	0	\emptyset
0	1	1	\emptyset	0
1	0	1	1	\emptyset
1	1	0	\emptyset	1

en la que hemos puesto " \emptyset " en los lugares donde es indiferente que sea "0" ó "1"; como ya sabemos, esto nos ayuda en la minimización del circuito. Las tablas de Karnaugh de J y K en función de e y q son:



de las que obtenemos

$$J = K = e$$

por lo que la realización con JK será la de la figura 3.13.

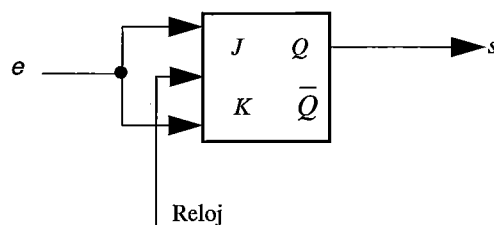


Figura 3.13.

6.2.2. Reconocedor de 010

El diagrama de Moore del AF reconocedor de la cadena 010 es el de la figura 2.12. Como hay cuatro estados necesitaremos dos biestables. Hagamos la siguiente asignación de estados: $q_1 = 00$; $q_2 = 01$; $q_3 = 10$; $q_4 = 11$. Con ello podemos escribir la tabla de transiciones en binario, y, al mismo tiempo, para cada transición, las entradas de los biestables; si éstos van a ser JK obtenemos:

e	q	s	z	J_1	K_1	J_2	K_2
0	00	0	01	0	\emptyset	1	\emptyset
0	01	0	01	0	\emptyset	\emptyset	0
0	10	0	11	\emptyset	0	1	\emptyset
0	11	1	01	\emptyset	1	\emptyset	0
1	00	0	00	0	\emptyset	0	\emptyset
1	01	0	10	1	\emptyset	\emptyset	1
1	10	0	00	\emptyset	1	0	\emptyset
1	11	1	10	\emptyset	0	\emptyset	1

Las cuatro últimas columnas con las tres primeras definen las tablas de excitaciones. Si llamamos Q_1 y Q_2 (salidas de los biestables) al primero y segundo dígito de q , minimizando por Karnaugh podemos hallar las ecuaciones lógicas de las entradas de los biestables en función de e , Q_1 y Q_2 ; el resultado es:

$$J_1 = e \cdot Q_2; K_1 = e \oplus Q_2; J_2 = \bar{e}; K_2 = e$$

Por otra parte, de las tres primeras columnas se tendrá la función de salida, es decir, s en función de e , Q_1 y Q_2 . Minimizando resulta:

$$s = Q_1 \cdot Q_2$$

Con lo cual podemos trazar el circuito de la figura 3.14.

6.2.3. Contador BCD módulo 10

Partimos del diagrama de Moore de la figura 2.17. Como hay diez estados necesitaremos cuatro biestables ($2^3 < 10 < 2^4$), que supondremos del tipo JK .

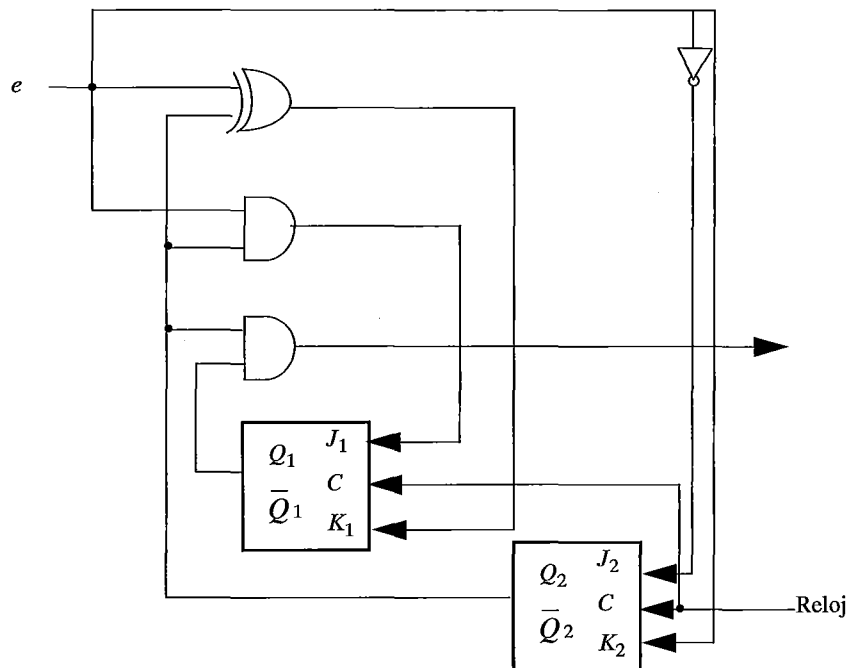


Figura 3.14.

Codifiquemos los estados con los mismos dígitos que las salidas asociadas:

$$q_0 = s_0 = 0000$$

$$q_1 = s_1 = 0001$$

$$q_2 = s_2 = 0010$$

:

$$q_9 = s_9 = 1001$$

De esta manera se simplifica al máximo el circuito que realiza la función de salida, puesto que será $s = q$.

Escribimos a continuación la tabla de transición junto con la de excitaciones:

e	Q_3	Q_2	Q_1	Q_0	Z	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0000	0	\emptyset	0	\emptyset	0	\emptyset	0	\emptyset
0	0	0	0	1	0001	0	\emptyset	0	\emptyset	0	\emptyset	\emptyset	0
0	0	0	1	0	0010	0	\emptyset	0	\emptyset	\emptyset	0	0	\emptyset

e	Q_3	Q_2	Q_1	Q_0	Z	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	1	1	0011	0	\emptyset	0	\emptyset	\emptyset	0	\emptyset	0
0	0	1	0	0	0100	0	\emptyset	\emptyset	0	0	\emptyset	0	\emptyset
0	0	1	0	1	0101	0	\emptyset	\emptyset	0	0	\emptyset	\emptyset	0
0	0	1	1	0	0110	0	\emptyset	\emptyset	0	\emptyset	0	0	\emptyset
0	0	1	1	1	0111	0	\emptyset	\emptyset	0	\emptyset	0	\emptyset	0
0	1	0	0	0	1000	\emptyset	0	0	\emptyset	0	\emptyset	0	\emptyset
0	1	0	0	1	1001	\emptyset	0	0	\emptyset	0	\emptyset	\emptyset	0
1	0	0	0	0	0001	0	\emptyset	0	\emptyset	0	\emptyset	1	\emptyset
1	0	0	0	1	0010	0	\emptyset	0	\emptyset	1	\emptyset	\emptyset	1
1	0	0	1	0	0011	0	\emptyset	0	\emptyset	\emptyset	0	1	\emptyset
1	0	0	1	1	0100	0	\emptyset	1	\emptyset	\emptyset	1	\emptyset	1
1	0	1	0	0	0101	0	\emptyset	\emptyset	0	0	\emptyset	1	\emptyset
1	0	1	0	1	0110	0	\emptyset	\emptyset	0	1	\emptyset	\emptyset	1
1	0	1	1	0	0111	0	\emptyset	\emptyset	0	\emptyset	0	1	\emptyset
1	0	1	1	1	1000	1	\emptyset	\emptyset	1	\emptyset	1	\emptyset	1
1	1	0	0	0	1001	\emptyset	0	0	\emptyset	0	\emptyset	1	\emptyset
1	1	0	0	1	0000	\emptyset	1	0	\emptyset	0	\emptyset	\emptyset	1

Tenemos así en forma de tablas de verdad J_i y K_i en función de e, Q_1, Q_2, Q_3, Q_4 . Obsérvese que no están todas las combinaciones de variables binarias de estado, debido a que sólo tenemos diez estados de los dieciséis posibles con cuatro variables. Para tales combinaciones podemos tomar cualquier valor (\emptyset) en las J y K . La tabla de Karnaugh de J_3 será la que ilustra el diagrama de la siguiente página, y de ella deducimos:

$$J_3 = e \cdot Q_2 \cdot Q_1 \cdot Q_0$$

Y, análogamente,

$$J_2 = e \cdot Q_1 \cdot Q_0;$$

$$J_1 = e \cdot \bar{Q}_3 \cdot Q_0;$$

$$J_0 = e$$

$$K_3 = e \cdot Q_0$$

$$K_2 = e \cdot Q_1 \cdot Q_0$$

$$K_1 = e \cdot Q_0$$

$$K_0 = e$$

Resulta así el circuito de la figura 3.15.

Q_1	Q_0	Q_3			
		Q_2			
		00	01	11	10
00		0	0	\emptyset	\emptyset
01		0	0	\emptyset	\emptyset
11		0	0	\emptyset	\emptyset
10		0	0	\emptyset	\emptyset

$$e = 0$$

		Q_3		Q_2	
		Q_1	Q_0		
		00	01	11	10
00		0	0	\emptyset	\emptyset
01		0	0	\emptyset	\emptyset
11		0	1	\emptyset	\emptyset
10		0	0	\emptyset	\emptyset

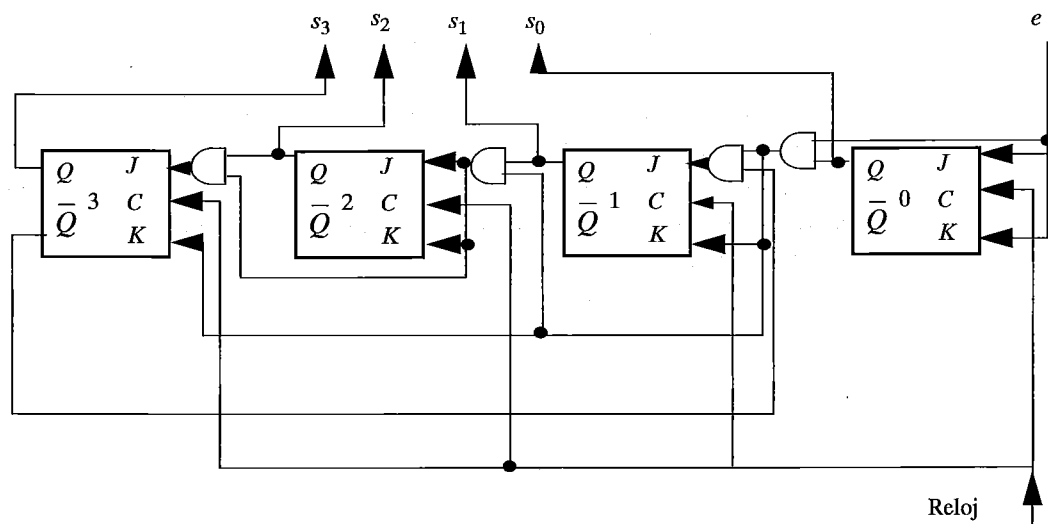
$$e = 1$$


Figura 3.15.

7. Resumen

Hemos estudiado algunos de los elementos utilizados como memorias en la realización física de los AF (circuitos secuenciales). Conociendo el funcionamiento de los elementos que lo componen, el análisis de un circuito secuencial se puede llevar a cabo deduciendo paso a paso el cronograma correspondiente a una determinada cadena, y también se puede obtener el diagrama de Moore, que tendrá, si p es el número de elementos binarios de memoria, 2^p estados.

Hemos visto el procedimiento general para la síntesis de circuitos secuenciales, ilustrándolo con algunos ejemplos. No hemos abordado, por su mayor complejidad, el diseño de máquinas incompletamente especificadas ni el problema de la asignación de estados.

8. Notas histórica y bibliográfica

Véase el apartado 11 correspondiente al capítulo 3 de la primera parte del libro, "Lógica".

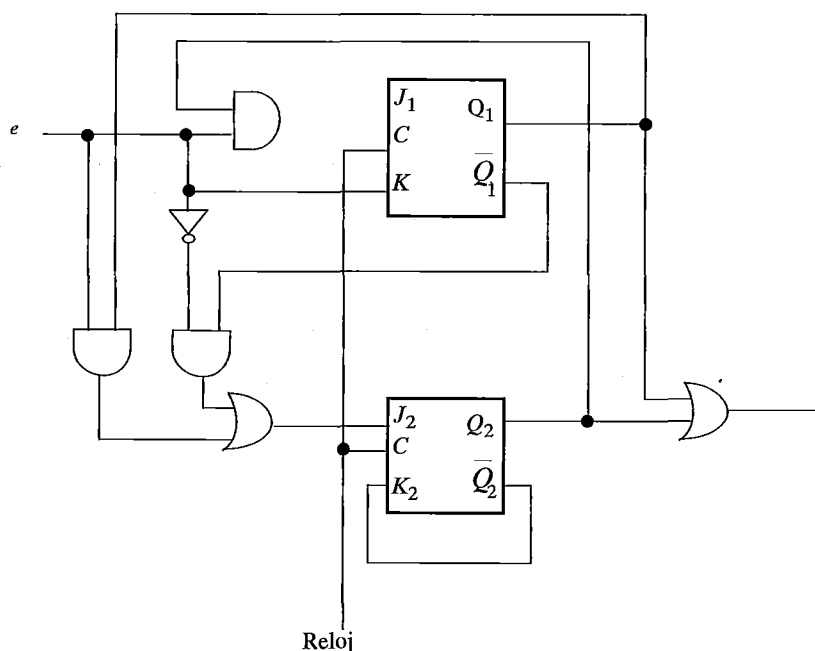
9. Ejercicios

- 9.1. Diseñar el sumador binario serie utilizando un biestable de tipo JK .
- 9.2. En el ejercicio 9 del capítulo 2, supóngase que los tres posibles símbolos de entrada se codifican en binario: $1 = 01$; $2 = 10$; $3 = 11$ (es decir, habrá dos hilos de entrada). Diseñar el circuito secuencial correspondiente.
- 9.3. Diseñar un contador módulo 10 en exceso de 3, utilizando JK . (Véase el ejercicio 12.4 del capítulo 3 del tema "Lógica".)
- 9.4. Diseñar contadores módulo 10 en BCD natural y en exceso de 3, con biestables tipo D .
- 9.5. Diseñar circuitos con biestables JK y D para un autómata retardador de dos intervalos ($s(t) = e(t - 2)$).
- 9.6. Diseñar un circuito que gobierne el funcionamiento de tres luces (verde, roja y amarilla) reguladoras de tráfico. La luz verde deberá permanecer encendida durante un período de 40 seg, con la luz roja apagada; al cabo de este tiempo la situación cambiará (verde apagada, roja encendida) durante otros 40 seg, y así sucesivamente. La luz amarilla solamente se encenderá (simultáneamente con la verde) durante los diez segundos que preceden al encendido de la roja. Para ello, el circuito tendrá tres salidas binarias (V, R, A) y una sola entrada consistente en impulsos de período 10 seg.

- 9.7. Diseñar un circuito secuencial que simule la situación descrita para el "castillo encantado" (véase el apartado 2.3 correspondiente al capítulo 2 de esta parte).
- 9.8. Dado el autómata definido por la tabla que hay a continuación, expresar con palabras su funcionamiento (es decir, cuándo da 0 ó 1 a la salida dependiendo de la secuencia de ceros y unos a la entrada), y diseñar un circuito para materializarlo.

	0	1
$q_1/0$	q_2	q_1
$q_2/0$	q_3	q_1
$q_3/1$	q_3	q_1

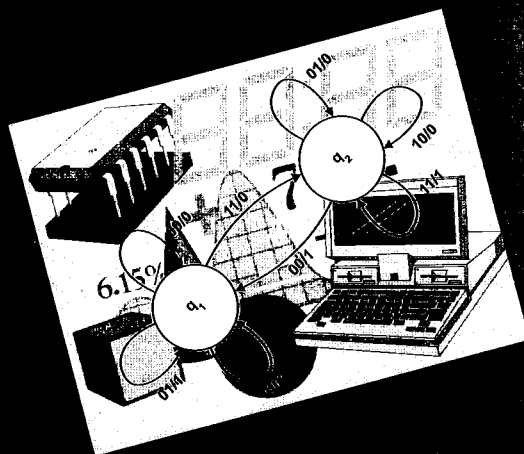
- 9.9. Analizar el circuito que ilustra la figura y comprobar si es posible diseñar otro que, realizando la misma función, sea más sencillo..



- 9.10. Realizar el análisis correspondiente a los diseños de los contadores BCD natural (véase apartado 6.2.3) y BCD exceso de 3 (ejercicios 3 y 4) para poder decir qué ocurre si, por causa de una perturbación exterior o de un fallo de funcionamiento, se pasa a alguno de los seis estados no permitidos

Fundamentos de informática

4



4

Autómatas reconocedores y lenguajes regulares

1. Reconocedor finito

1.1. Definición

Hemos definido al principio del libro un *lenguaje*, L , sobre un alfabeto, A , como un subconjunto cualquiera de A^* . En este capítulo vamos a ver que ciertos lenguajes pueden asociarse con autómatas finitos que sirven como reconocedores de las cadenas pertenecientes a tales lenguajes.

Un reconocedor finito de un lenguaje L es un AF que sólo acepta las cadenas de dicho lenguaje, en el sentido de que, inicializado en un estado predeterminado, q_1 , si se le introduce una cadena de entrada $x_1 \in L$, da un símbolo final de salida que corresponde a "aceptación" (por ejemplo, $s = 1$), mientras que para $x_1 \notin L$ produce una salida de "no aceptación" (por ejemplo $s = 0$). En adelante supon-

dremos siempre que hablemos de reconocedores que se trata de máquinas de Moore. Podemos entonces hacer abstracción del alfabeto de salida y de la función g , y considerar el subconjunto de estados $F \subset Q$ que producen la salida de aceptación, a los que llamaremos estados finales. De acuerdo con esto, daremos la siguiente definición formal.

Un *reconocedor finito* es una quintupla

$$R = \langle E, Q, f, q_1, F \rangle$$

donde:

E es un conjunto finito (alfabeto de entrada).

Q es un conjunto finito (conjunto de estados).

f es una función $f: E \times Q \rightarrow Q$ (función de transición).

$q_1 \in Q$ es un estado designado como estado inicial.

$F \subset Q$ es un conjunto de estados designados como estados finales.

Llamaremos $L(R)$ al conjunto de cadenas aceptadas por R , es decir,

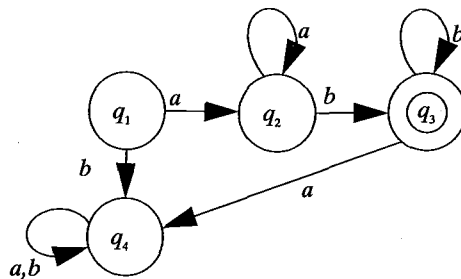
$$L(R) = \{x \in E^* \mid f(x, q_1) \in F\}$$

(con dominio de f ampliado a E^*).

Seguiremos el convenio de representar en los diagramas de Moore los estados de aceptación con círculo doble, y en las tablas de transición, encerrados en un círculo.

1.2. Ejemplos

Ejemplo 1.2.1

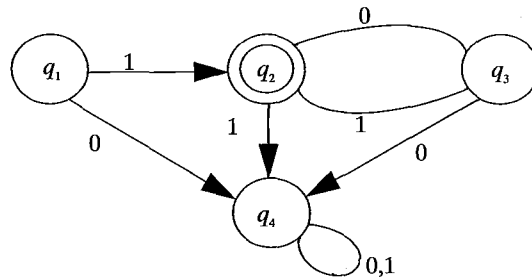


	a	b
q ₁	q ₂	q ₄
q ₂	q ₂	q ₃
q ₃	q ₄	q ₃
q ₄	q ₄	q ₄

Figura 4.1.

$$L(R_1) = \{ab, aab, \dots, abb, abbb, \dots, aabb, \dots\} = \{a^n b^m \mid n \geq 1, m \geq 1\}$$

Ejemplo 1.2.2

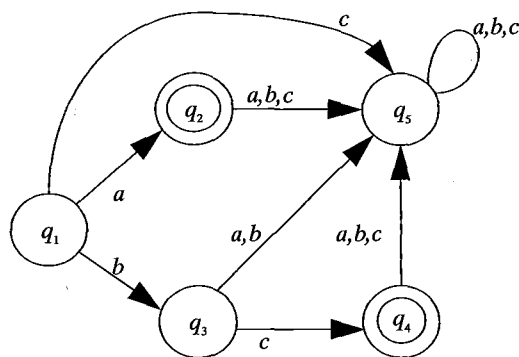


	0	1
q_1	$q_4 q_2$	
q_2	$q_3 q_4$	
q_3	$q_4 q_2$	
q_4	$q_4 q_4$	

Figura 4.2.

$$L(R_2) = \{1, 101, 10101, \dots\} = \{1(01)^n \mid n \geq 0\}$$

Ejemplo 1.2.3

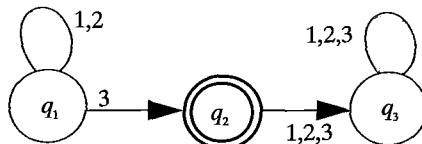


	a	b	c
q_1	q_2	q_3	q_5
q_2	q_5	q_5	q_5
q_3	q_5	q_5	q_4
q_4	q_5	q_5	q_5
q_5	q_5	q_5	q_5

Figura 4.3.

$$L(R_3) = \{a, bc\}$$

Ejemplo 1.2.4



	1	2	3
q_1	q_1	$q_1 q_2$	
q_2	q_3	$q_3 q_3$	
q_3	q_3	$q_3 q_3$	

Figura 4.4.

$$L(R_4) = \{3, 13, 113, 1113, \dots, 123, 1223, \dots, 213, 2113, \dots\} = \{1^{n_1} 2^{n_2} 1^{n_3} 2^{n_4} \dots 3 \mid n_1, n_2, \dots \geq 0\}$$

2. Lenguajes aceptados por reconocedores finitos

2.1. Planteamiento del problema

Podemos preguntarnos si, dado un lenguaje cualquiera, $L \subset E^*$, podemos siempre encontrar un reconocedor finito, R , tal que $L(R) = L$. Observemos que esa pregunta ya la habíamos dejado planteada, en términos más generales, en el capítulo 2, cuando, al final del apartado 3.1, decíamos: dada una máquina ¿podemos encontrar su circuito?

Baste un ejemplo para demostrar que existen lenguajes a los que no corresponde ningún reconocedor finito. Consideremos $E = \{0, 1\}$, y sea

$$L = \{1^{n^2} \mid n \geq 1\}.$$

Supongamos que existe un R , con p estados, tal que $L(R) = L$. Para cadenas del tipo $x = 1^i$ tendremos que $f(1^i, q_1) \in F$ si i es un cuadrado perfecto. Para las $p + 1$ cadenas $x_0 = 1^0, x_1 = 1^1, \dots, x_p = 1^p$, tendremos como estados resultantes $f(x_0, q_1), f(x_1, q_1), \dots, f(x_p, q_1)$, pero como el reconocedor sólo tiene p estados, al menos dos de estos estados resultantes deberán ser el mismo: $f(1^i, q_1) = f(1^j, q_1)$, con $j - i \leq p$. Entonces, si el reconocedor acepta

$$1^{n^2}$$

también deberá aceptar

$$1^{n^2 + (j-i)},$$

ya que

$$\begin{aligned} f(1^{n^2}, q_1) &= f(1^i 1^{n^2-i}, q_1) = f[1^{n^2-i}, f(1^i, q_1)] = \\ &= f[1^{n^2-i}, f(1^j, q_1)] = f(1^j 1^{n^2-i}, q_1) = f(1^{n^2 + (j-i)}, q_1). \end{aligned}$$

Ahora bien, siempre podemos tomar n suficientemente grande como para que $(n+1)^2 - n^2 > p$, y, por tanto $(n+1)^2 - n^2 > (j-i)$, con lo que $n^2 + (j-i) < (n+1)^2$ no será un cuadrado perfecto, y el reconocedor responderá incorrectamente al aceptar

$$1^{n^2 + (j-i)}.$$

Vemos pues que *los lenguajes aceptados por algún reconocedor finito son un subconjunto de todos los lenguajes posibles sobre E^* .*

Ahora podemos preguntarnos: ¿existe alguna propiedad que caracterice a los lenguajes que son aceptados por un reconocedor finito? Si tal propiedad existe parece lógico pensar que pueda deducirse de la capacidad del reconocedor para responder de distinto modo a diferentes cadenas de entrada.

2.2. Relación equirrespuesta de un reconocedor finito

En el capítulo 2 definimos de una manera general el comportamiento de entrada-estados de un AF como un homomorfismo $K: \langle E^*, > \rightarrow \langle Q^0, \circ \rangle$, tal que a cada $x \in E^*$ corresponde una transformación entre estados, $K(x): Q \rightarrow Q$, y esto nos permitió definir una relación equirrespuesta en E^* tal que $x \equiv y$, si y sólo si $K(x) = K(y)$, es decir, las cadenas x e y estarán en relación si y sólo si producen la misma transformación entre estados:

$$(\forall q_i \in Q) [(x \equiv y) \leftrightarrow f(x, q_i) = f(y, q_i)].$$

Véamos que esta relación es una relación de congruencia en $\langle E^*, >$ y particiona E^* en un número finito de clases de equivalencia (es decir, es de índice finito, menor o igual a n , siendo n el número de estados).

En el caso del reconocedor finito, el estado inicial, q_1 , es fijo, y el comportamiento de entrada-estados será más bien una función $K_R: E^* \rightarrow Q$ que aplica a cada $x \in E^*$ un $q \in Q$ de tal manera que $K_R(x) = f(x, q_1)$. Podemos igualmente definir una relación equirrespuesta, \equiv_R en E^* :

$$(\forall x, y \in E^*) [(x \equiv_R y) \leftrightarrow (f(x, q_1) = f(y, q_1))].$$

Esta relación, como es fácil ver, es una relación de equivalencia. Además $(x \equiv y) \rightarrow (x \equiv_R y)$, pero no a la inversa (se dice que \equiv refina a \equiv_R), por lo que en la partición de E^* inducida por \equiv_R habrá un número igual o inferior de clases de equivalencia que en la inducida por \equiv .

Si bien \equiv es una relación de congruencia en $\langle E^*, >$, \equiv_R sólo es una relación de congruencia derecha. Esto quiere decir que $(x \equiv_R y) \rightarrow (xz \equiv_R yz)$, pero en general, no es cierto que $(zx \equiv_R zy)$. En efecto, si llamamos $q_i = f(x, q_1) = f(y, q_1)$ y $q_j = f(z, q_1)$, tendremos: $f(xz, q_1) = f(z, f(x, q_1)) = f(z, q_i) = q_j$ y análogamente $f(yz, q_1) = q_j$; sin embargo, $f(zx, q_1) = f(x, f(z, q_1))$, que en general será diferente de $f(y, f(z, q_1))$.

Así pues, a cada reconocedor finito corresponde una partición de E^* en clases de equivalencia tal que si dos cadenas están en la misma clase ambas cadenas conducen al mismo estado. Además, esta relación de equivalencia es de índice finito (menor o igual que n , siendo n el número de estados del reconocedor) y es

una relación de congruencia derecha en E^* . Veamos cómo pueden aplicarse estas conclusiones para caracterizar a los lenguajes aceptados por reconocedores finitos.

2.3. Condición para que un lenguaje sea aceptado por un reconocedor finito

Dado un lenguaje $L \subset E^*$, definimos la *relación de congruencia derecha inducida por L* , \cong_L , así:

$$(\forall x, y, z \in E^*) [(x \cong_L y) \leftrightarrow (xz \in L \leftrightarrow yz \in L)]$$

Es evidente que se trata de una relación de equivalencia. Para ver que también es una congruencia derecha basta suponer que $z = z_1 z_2$, con lo que

$$(\forall x, y, z_1, z_2 \in E^*) [(x \cong_L y) \leftrightarrow (xz_1 z_2 \in L \leftrightarrow yz_1 z_2 \in L) \leftrightarrow (xz_1 \cong_L yz_1)]$$

Vamos a demostrar ahora la principal conclusión de este apartado 2:
 $L \subset E^*$ es un lenguaje aceptado por un reconocedor finito si y sólo si la relación de congruencia derecha inducida por L tiene índice finito.

Vemos primero que si $L = L(R)$, entonces \cong_L tiene índice finito:

a) Por definición de \cong_R

$$(x \cong_R y) \leftrightarrow (f(x, q_1) = f(y, q_1))$$

b) Por ser \cong_R una congruencia derecha,

$$(x \cong_R y) \rightarrow [f(xz, q_1) = f(yz, q_1)]$$

c) Es claro que

$$[f(xz, q_1) = f(yz, q_1)] \rightarrow [(f(xz, q_1) \in F) \leftrightarrow (f(yz, q_1) \in F)]$$

d) Y también que

$$[(f(xz, q_1) \in F) \leftrightarrow (f(yz, q_1) \in F)] \leftrightarrow [(xz \in L) \leftrightarrow (yz \in L)]$$

e) Por definición de \cong_L ,

$$[(xz \in L) \leftrightarrow (yz \in L)] \leftrightarrow (x \cong_L y)$$

- f) El razonamiento constituido por las premisas b), c), d) y e) nos lleva a la conclusión de que

$$(x \equiv_R y) \rightarrow (x \equiv_L y)$$

$(\equiv_R \text{ refina a } \equiv_L)$. Por consiguiente, el índice (número de clases de equivalencia) de la partición inducida en E^* por \equiv_L es igual o inferior al de la partición inducida por \equiv_R , y como ésta es de índice finito, \equiv_L también lo será.

A la inversa, supongamos que L es un lenguaje que induce una relación de congruencia derecha en E^* que es de índice finito. Vamos a construir un reconocedor finito, R_L , tal que $L(R_L) = L$. Llamemos $[x]$ a la clase de equivalencia de E^*/\equiv_L que contiene a $x \in E^*$. Entonces definimos

$$R_L = \langle E, Q_L, f_L, q_1, F_L \rangle,$$

donde

$$Q_L = E^* / \equiv_L = \{[x]\} \text{ (finito, puesto que } \equiv_L \text{ es de índice finito)}$$

$$(\forall y \in E^*) (f_L(y, [x]) = [xy])$$

$$q_1 = [\lambda]$$

$$F_L = \{[x] \mid x \in L\}$$

(F_L está bien definida, pues si $x_1 \in L$ y $x_2 \in L$, según la definición de \equiv_L , haciendo $z = \lambda$ vemos que $x_1 \equiv_L x_2$, es decir, $[x_1] = [x_2]$).

El lenguaje aceptado por este reconocedor será:

$$\begin{aligned} L(R_L) &= \{y \in E^* \mid f_L(y, q_1) \in F_L\} = \{y \mid f_L(y, [\lambda]) \in F_L\} = \\ &= \{y \mid [y] \in F_L\} = \{y \mid y \in L\} = L \end{aligned}$$

Además, R_L está en forma mínima. En efecto, si $q_{L_1} = [x_1]$ fuera equivalente a $q_{L_2} = [x_2]$, esto querría decir que

$$(\forall y \in E^*) ((f_L(y, [x_1]) \in L) \leftrightarrow (f_L(y, [x_2]) \in L))$$

y por la definición de f_L ,

$$(\forall y \in E^*) (([x_1, y] \in L) \leftrightarrow ([x_2, y] \in L))$$

es decir, $x_1 \equiv_L x_2$; x_1 y x_2 se encuentran en la misma clase de equivalencia, por lo que $q_{L_1} = q_{L_2}$.

3. Conjuntos regulares y expresiones regulares

3.1. Los problemas de análisis y de síntesis

Acabamos de ver una condición necesaria y suficiente para que un lenguaje sea aceptado por un reconocedor finito. El problema de análisis consiste en deducir el lenguaje asociado a un determinado reconocedor, y el de síntesis en encontrar un reconocedor cuyo lenguaje sea un lenguaje dado. Ambos problemas los tenemos en teoría resueltos. En efecto, para el análisis, basta enumerar las cadenas que son aceptadas, como vimos en los ejemplos de 1.2; para la síntesis, hay que encontrar las clases de equivalencia de la relación de congruencia derecha inducida por L y construir el reconocedor como se ha indicado más arriba. El inconveniente está en que, al poder ser L un conjunto infinito, no es fácil trabajar con él, y no siempre podemos representarlo de una manera condensada, como hacíamos en los ejemplos de 1.2; además, salvo en el caso de que L sea finito (como el ejemplo 1.2.3), no tenemos un algoritmo para encontrar las clases de equivalencia inducidas por L . En este apartado vamos a exponer una herramienta, las expresiones regulares, especialmente introducida para trabajar con los lenguajes aceptados por reconocedores finitos, y que nos permitirá llegar a algoritmos para resolver los problemas de análisis y síntesis.

3.2. Conjuntos regulares

En primer lugar vamos a definir tres operaciones en el conjunto $\{L_1, L_2, \dots\}$ de subconjuntos de E^* (lenguajes sobre E):

a) Unión:

$$L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$$

b) Concatenación:

$$L_1 L_2 = \{x_1 x_2 \mid x_1 \in L_1 \wedge x_2 \in L_2\}$$

c) Cierre u operación estrella:

$$L^* = \{\lambda\} \cup \{L\} \cup \{LL\} \cup \{LLL\} \dots = \bigcup_{n=0}^{\infty} L^n$$

Decimos que un subconjunto de E^* , $L_R \subset E^*$, es regular si y sólo si:

- a) L_R es un subconjunto finito de E^* (puede ser $L = \emptyset$), o bien;
- b) L_R puede obtenerse a partir de subconjuntos finitos de E^* mediante un número finito de operaciones de unión, concatenación y cierre.

3.3. Expresiones regulares

Las expresiones regulares se introducen para describir los conjuntos regulares, y como éstos son lenguajes, las expresiones regulares serán metalenguajes.

Seguiremos el convenio de designar por $|\alpha|$ al conjunto descrito por la expresión regular α .

Definimos ahora el conjunto de expresiones regulares sobre un alfabeto $E = \{e_1, \dots, e_n\}$ y las operaciones suma, concatenación y cierre de la siguiente manera recursiva:

- a) λ , \emptyset y e_i ($i = 1, \dots, n$) son expresiones regulares tales que
 $|\lambda| = \{\lambda\}$; $|\emptyset| = \{\emptyset\}$ y $|e_i| = \{e_i\}$ ($i = 1, \dots, n$);¹
- b) si α y β son expresiones regulares, $\alpha + \beta$ es una expresión tal que
 $|\alpha + \beta| = |\alpha| \cup |\beta|$;
- c) si α y β son expresiones regulares, $\alpha \beta$ es una expresión regular tal que
 $|\alpha \beta| = |\alpha| |\beta|$;
- d) si α es una expresión regular, α^* es una expresión regular tal que
 $|\alpha^*| = |\alpha|^*$.

Como estas operaciones corresponden a las utilizadas para definir los conjuntos regulares, a todo conjunto corresponderá al menos una expresión regular. Veamos algunos ejemplos

E	Expresión regular, α	Conjunto regular, $ \alpha $
1. $\{a, b\}$	aa^*bb^*	Conjunto de todas las cadenas de E^* constituidas por "a" seguido de "a" cualquier número de veces (o ninguna), seguido de "b" y seguido de "b" cualquier número de veces (o ninguna).

¹ λ es la cadena vacía (elemento neutro para la concatenación de cadenas), y \emptyset es el conjunto vacío (elemento neutro para la unión de conjuntos).

	E	Expresión regular, α	Conjunto regular, $ \alpha $
2.	$\{0, 1\}$	$1(01)^*$	Conjunto de cadenas que empiezan por "1" y sigue (01) cualquier número de veces (o ninguna).
3.	$\{a, b, c\}$	$a + bc$	$\{a, bc\}$.
4.	$\{1, 2, 3\}$	$(1 + 2)^*3$	Conjunto de cadenas formadas mediante los símbolos 1 y 2 sucediéndose cualquier número de veces (y en cualquier orden), y siempre dando por finalizada la cadena mediante el símbolo 3.
5.	$\{e_1, e_2, \dots, e_n\}$	$(e_1 + e_2 + \dots + e_n)^*$	E^* .
6.	$\{0, 1\}$	$(01)^*$	El conjunto formado por λ y todas las cadenas constituidas por la cadena 01 repetida cualquier número de veces.
7.	$\{0, 1\}$	0^*10^*	Conjunto de todas las cadenas que tienen un "1" (y sólo uno).

Obsérvese que los cuatro primeros ejemplos corresponden exactamente a los cuatro ejemplos del apartado 1.2.

Dos expresiones regulares son iguales si designan al mismo conjunto regular:

$$(\alpha = \beta) \leftrightarrow |\alpha| = |\beta|$$

Teniendo esto presente, es fácil demostrar las siguientes propiedades de las expresiones regulares:

1. Asociatividad de la concatenación:

$$\alpha(\beta\gamma) = (\alpha\beta)\gamma$$

2. Distributividad de la suma:

$$\alpha\beta + \alpha\gamma = \alpha(\beta + \gamma);$$

$$\beta \alpha + \gamma \alpha = (\beta + \gamma) \alpha$$

3. \emptyset es elemento neutro para la suma:

$$\alpha + \emptyset = \emptyset + \alpha = \alpha$$

4. \emptyset es un cero para la concatenación:

$$\alpha \emptyset = \emptyset \alpha = \emptyset$$

5. λ es elemento neutro para la concatenación:

$$\alpha \lambda = \lambda \alpha = \alpha$$

6. Propiedades de la operación cierre:

$$a) (\alpha + \beta)^* = (\alpha^* + \beta^*)^* = (\alpha^* \beta^*)^*$$

$$b) (\alpha + \lambda)^* = \alpha^* + \lambda = \alpha^*$$

$$c) \alpha \alpha^* + \lambda = \alpha^*$$

$$d) \lambda^* = \emptyset^* = \lambda$$

Por ejemplo, para demostrar que $(\alpha + \beta)^* = (\alpha^* \beta^*)^*$ tendremos en cuenta que:

$$\begin{aligned} |(\alpha + \beta)^*| &= \{\lambda\} \cup |\alpha + \beta| \cup |\alpha + \beta|^2 \cup \dots = \\ &= \{\lambda\} \cup |\alpha| \cup |\beta| \cup |\alpha|^2 \cup |\beta|^2 \cup |\alpha||\beta| \cup |\beta||\alpha| \cup \dots \end{aligned}$$

y, por otra parte,

$$\begin{aligned} |(\alpha^* \beta^*)^*| &= \{\lambda\} \cup |\alpha^* \beta^*| \cup |\alpha^* \beta^*|^2 \cup \dots = \\ &= \{\lambda\} \cup |\alpha^*||\beta^*| \cup |\alpha^*||\beta^*||\alpha^*||\beta^*| \cup \dots = \\ &= \{\lambda\} \cup |\alpha| \cup |\beta| \cup |\alpha|^2 \cup |\beta|^2 \cup |\alpha||\beta| \cup |\beta||\alpha| \cup \dots \end{aligned}$$

Pasemos ahora a ver que *todos los lenguajes aceptados por reconocedores finitos son conjuntos regulares y que todo conjunto regular es un lenguaje aceptado por un reconocedor finito, lo que nos va a permitir resolver los problemas de análisis y de síntesis, respectivamente.*

4. Resolución de los problemas de análisis y de síntesis de un reconocedor finito

4.1. Análisis

4.1.1. Teorema de análisis

Todo lenguaje aceptado por un reconocedor finito es un conjunto regular. Supongamos que el reconocedor finito tiene n estados, $Q = \{q_1, \dots, q_n\}$, con estado inicial q_1 y con estados finales $F = \{q_{f_1}, q_{f_2}, \dots, q_{f_r}\}$ ($n \geq r \geq 0$). El lenguaje aceptado es:

$$L(R) = \{x \mid f(x, q_1) \in F\} = \cup \{R_{1j} \mid q_j \in F\}$$

con $R_{ij} = \{x \mid f(x, q_i) = q_j\}$ (conjunto de cadenas que llevan del estado q_i al estado q_j). Basta entonces demostrar que los R_{ij} son conjuntos regulares.

Vamos a definir R_{ij}^k ($0 \leq k \leq n$) como el conjunto de cadenas que llevan de q_i a q_j sin pasar por ningún q_l tal que $l > k$. En particular R_{ij}^0 serán las cadenas que llevan de q_i a q_j sin pasar por ningún otro estado, por lo que son símbolos, es decir, es un subconjunto de $E \cup \{\lambda\}$ y, por tanto, es regular. Supongamos que R_{ij}^{k-1} es regular para todo i y j y $k \geq 1$ y demostremos que entonces R_{ij}^k es regular. En efecto, basta comprobar que

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} (R_{kk}^{k-1})^* R_{kj}^{k-1}$$

Entonces, R_{ij}^k es regular para todo k , y, en particular, $R_{ij} = R_{ij}^n$ es regular.

4.1.2. Algoritmo de análisis

Como corolario del teorema anterior se desprende un procedimiento para obtener la expresión regular del lenguaje aceptado por un determinado reconocedor. En efecto, llamemos α_{ij}^k a la expresión regular que designa al conjunto R_{ij}^k : $|\alpha_{ij}^k| = R_{ij}^k$; entonces, si q_1 es el estado inicial, la expresión regular de $L(R)$ será:

$$\begin{cases} \alpha_{1f_1} + \alpha_{1f_2} + \dots + \alpha_{1f_r}, & \text{si } r > 0 \\ \emptyset & \text{si } r = 0 \end{cases}$$

Los α_{ij} se calcularán recursivamente teniendo en cuenta los conjuntos a los que representan:

$$\begin{aligned} |\alpha_{ij}^0| &= \{e \in E \cup \{\lambda\} \mid f(e, q_i) = q_j\} \\ \alpha_{ij}^k &= \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1} (\alpha_{kk}^{k-1})^* \alpha_{kj}^{k-1}, \quad 0 < k \leq n \\ \alpha_{ij} &= \alpha_{ij}^n \end{aligned}$$

4.1.3. Ejemplos

Vamos a tomar los mismos cuatro ejemplos del apartado 1.2: partimos de los diagramas y debemos llegar a las expresiones regulares que ya conocemos.

1. Sólo hay un estado final, q_3 , luego la expresión regular será:

$$\alpha_{13} = \alpha_{13}^4$$

$$\alpha_{13}^4 = \alpha_{13}^3 + \alpha_{14}^3 (\alpha_{44}^3)^* \alpha_{43}^3$$

Pero $\alpha_{43}^3 = \emptyset$, como se ve directamente en el diagrama de Moore, por lo que

$$\alpha_{13}^4 = \alpha_{13}^3 = \alpha_{13}^2 + \alpha_{13}^2 (\alpha_{33}^2)^* \alpha_{33}^2$$

Calculemos pues, α_{13}^2 y α_{33}^2 :

$$\alpha_{13}^2 = \alpha_{13}^1 + \alpha_{12}^1 (\alpha_{22}^1)^* \alpha_{23}^1$$

$$\alpha_{13}^1 = \alpha_{13}^0 + \alpha_{11}^0 (\alpha_{11}^0)^* \alpha_{13}^0 = \emptyset + \lambda (\lambda)^* \emptyset = \emptyset$$

(también se ve directamente en el diagrama)

$$\alpha_{12}^1 = \alpha_{12}^0 + \alpha_{11}^0 (\alpha_{11}^0)^* \alpha_{12}^0 = a + \lambda (\lambda)^* a = a$$

$$\alpha_{22}^1 = \alpha_{22}^0 + \alpha_{21}^0 (\alpha_{11}^0)^* \alpha_{12}^0 = a + \emptyset (\lambda)^* a = a$$

$$\alpha_{23}^1 = \alpha_{23}^0 + \alpha_{21}^0 (\alpha_{11}^0)^* \alpha_{13}^0 = b + \emptyset (\lambda)^* \emptyset = b$$

Luego:

$$\alpha_{13}^2 = \emptyset + a(a^*)b = aa^*b$$

Podríamos calcular α_{33}^2 de manera análoga, pero ya se ve directamente en el diagrama que saldrá $\alpha_{33}^2 = b^*$. Por tanto,

$$\alpha_{13} = \alpha_{13}^4 = \alpha_{13}^3 = aa^*b + aa^*b(b^*)^*b^* =$$

$$= aa^*b + aa^*bb^* = aa^*b(\lambda + b^*) = aa^*bb^*$$

2. Como sólo hay un estado final, q_2 , la expresión regular será:

$$\alpha_{12} = \alpha_{12}^4 = \alpha_{12}^3 + \alpha_{14}^3 (\alpha_{44}^3)^* \alpha_{42}^3$$

Se ve sobre el diagrama que $\alpha_{42}^3 = \emptyset$, por lo que ya no tenemos necesidad de calcularla, ni de calcular α_{14}^3 ni α_{44}^3 :

$$\alpha_{12}^2 = \alpha_{12}^3 = \alpha_{12}^2 + \alpha_{13}^2 (\alpha_{33}^2)^* \alpha_{32}^2$$

Omitimos los cálculos de α_{12}^2 , α_{13}^2 , α_{33}^2 y α_{32}^2 , que se hacen por el mismo procedimiento de reducción, resultando:

$$\alpha_{12}^2 = 1; \alpha_{13}^2 = 10; \alpha_{33}^2 = 10; \alpha_{32}^2 = 1$$

(En los cálculos intermedios resulta una expresión de la forma \emptyset^* ; no olvidar que $\emptyset^* = \lambda$, no \emptyset). Sustituyendo y aplicando las propiedades de las operaciones,

$$\alpha_{12}^2 = 1 + 10(10)^*1 = (\lambda + 10(10)^*)1 = (10)^*1 = 1(01)^*$$

Dejamos como ejercicio la obtención de las expresiones regulares de los otros dos ejemplos. (En el ejemplo 1.2.3 hay dos estados finales, q_2 y q_4 , por lo que la expresión será $\alpha_{12}^5 + \alpha_{14}^5 = \alpha_{12}^5 + \alpha_{14}^5 = \dots$).

Como se habrá observado, el procedimiento es bastante engorroso para aplicarlo manualmente, pero es un algoritmo general que puede programarse para su ejecución automática. En la ejecución manual a veces puede simplificarse intuitivamente; así, cuando decíamos, en el ejemplo 1.2.1, que se ve directamente en el diagrama que $\alpha_{43}^3 = \emptyset$, lo que nos evita muchos cálculos. De hecho, si el diagrama no es muy complicado, es preferible obtener la expresión regular por simple inspección.

4.2. Síntesis

4.2.1. Teorema de síntesis

Todo conjunto regular es un lenguaje aceptado por un reconocedor finito. Hay publicadas varias demostraciones de este teorema, bastante laboriosas todas ellas, por lo que, a fin de no alargar excesivamente este tema, nos permitimos no incluir ninguna, remitiendo al lector a las notas bibliográficas del apartado 6. En cambio, nos parece interesante dar un algoritmo que permite obtener directamente el reconocedor de un conjunto regular denotado por su expresión regular. Para ello necesitamos introducir un nuevo concepto: el de derivadas de una expresión regular.

4.2.2. Derivadas de una expresión regular

Consideremos una expresión regular, α , que designa a un conjunto regular L_R , $|\alpha| = L_R$, y consideremos el subconjunto de L_R formado por todas las cadenas de

L_R que empiezan por un determinado símbolo, e . Definimos el *cociente izquierdo* de L_R por e , $L_R \setminus e$, como el conjunto resultante de suprimir e en todas esas cadenas:

$$L_R \setminus e = \{x \mid ex \in L_R\}$$

y definimos la *derivada de α respecto al símbolo e* , $D_e(\alpha)$, como la expresión regular de $L_R \setminus e$. (Es fácil ver que si L_R es regular, $L_R \setminus e$ también lo es).

Por ejemplo, sea $\alpha = abc + d + a^*c$, es decir, $L_R = |\alpha| = \{abc, d, c, ac, aac, \dots\}$. Entonces,

$$L_R \setminus a = \{bc, c, ac, \dots\}; L_R \setminus b = \emptyset; L_R \setminus c = \lambda; L_R \setminus d = \lambda,$$

y las derivadas serán las respectivas expresiones regulares:

$$D_a(\alpha) = bc + a^*c; D_b(\alpha) = \emptyset; D_c(\alpha) = \lambda; D_d(\alpha) = \lambda.$$

Veamos algunas propiedades de las derivadas así definidas. De la definición es inmediato comprobar que:

$$a) \quad D_{e_1}(e_2) = \begin{cases} \lambda & \text{si } e_1 = e_2 \\ \emptyset & \text{si } e_1 \neq e_2 \end{cases}$$

$$b) \quad D_e(\lambda) = D_e(\emptyset) = \emptyset$$

$$c) \quad D_e(\alpha + \beta) = D_e(\alpha) + D_e(\beta)$$

Ya no es tan inmediato calcular la derivada de la concatenación de dos expresiones regulares ni la del cierre de una expresión regular.

Supongamos que $\gamma = \alpha\beta$. Si $|\alpha|$ no contiene la cadena vacía, λ , al suprimir "e" en las cadenas de $|\alpha\beta|$ resultarán las cadenas derivadas de α (es decir, todas aquellas que comiencen por "e", suprimiendo "e") concatenadas con las cadenas de β : $D_e(\alpha\beta) = D_e(\alpha)\beta$. Pero si $\lambda \in |\alpha|$, entonces $|\alpha\beta|$ contiene también a todas las cadenas de β , por lo que habrá que añadir $D_e(\beta)$. Si introducimos la expresión $\delta(\alpha)$ tal que

$$\delta(\alpha) = \emptyset \text{ si } \lambda \notin |\alpha|$$

$$\delta(\alpha) = \lambda \text{ si } \lambda \in |\alpha|$$

entonces podemos representar ambos casos en una sola expresión:

$$d) \quad D_e(\alpha\beta) = [D_e(\alpha)]\beta + \delta(\alpha) D_e(\beta)$$

Finalmente, veamos cómo se calcula la derivada de la operación cierre. Sabemos que

$$|\alpha^*| = |\alpha|^* = \{\lambda\} \cup |\alpha| \cup |\alpha||\alpha| \cup |\alpha||\alpha||\alpha| \cup \dots$$

por lo que

$$\begin{aligned} D_e(\alpha^*) &= D_e(\lambda) + D_e(\alpha) + D_e(\alpha\alpha) + D_e(\alpha\alpha\alpha) + \dots \\ D_e(\lambda) &= \emptyset \end{aligned}$$

Si $\lambda \notin |\alpha|$

$$\begin{aligned} D_e(\alpha^*) &= D_e(\alpha) + [D_e(\alpha)]\alpha + [D_e(\alpha)]\alpha\alpha + \dots = \\ &= D_e(\alpha) [\lambda + \alpha + \alpha\alpha + \dots] = [D_e(\alpha)]\alpha^* \end{aligned}$$

Si $\lambda \in |\alpha|$ llegamos al mismo resultado. En efecto:

$$D_e(\alpha^*) = D_e(\alpha) + [D_e(\alpha)]\alpha + D_e(\alpha) + [D_e(\alpha)]\alpha\alpha + D_e(\alpha\alpha) + \dots$$

que se reduce a la misma expresión anterior teniendo en cuenta la idempotencia de la suma. Luego:

$$e) \quad D_e(\alpha^*) = [D_e(\alpha)]\alpha^*$$

Teniendo en cuenta estas cinco propiedades se puede calcular la derivada de cualquier expresión regular sin tener que formar previamente el conjunto cociente.

Hemos visto la derivación respecto de un símbolo, $e \in E$; la operación se puede extender a derivación respecto a una cadena definiendo:

$$\begin{aligned} D_\lambda(\alpha) &= \alpha \\ D_{xe}(\alpha) &= D_e[D_x(\alpha)] \end{aligned}$$

Se puede demostrar (por inducción sobre la longitud de las cadenas) que el conjunto de derivadas diferentes de una expresión regular, es decir, $\{D_x(\alpha) \mid x \in E^*\}$ es finito.

4.2.3. Algoritmo de síntesis

Para construir un reconocedor en forma mínima del lenguaje denotado por la expresión regular α , se puede seguir el siguiente procedimiento:

1. Calcular $\{D_x(\alpha) \mid x \in E^*\}$.
2. El estado inicial es $q_1 = \alpha$; los otros son las diferentes $D_x(\alpha)$.
3. La función de transición es $f(e, \alpha) = D_e(\alpha)$; $f(e, D_e(\alpha)) = D_{xe}(\alpha)$.
4. El conjunto de estados finales es $F = \{D_x(\alpha) \mid \lambda \in |D_x(\alpha)|\}$.

4.2.4. Ejemplos

Para ilustrar la aplicación del algoritmo anterior vamos a tratar los mismos cuatro ejemplos del apartado 1.2. Teníamos allí cuatro reconocedores; las expresiones regulares de los correspondientes lenguajes las obtuvimos en los apartados 3.3 y 4.1.3. Pues bien, ahora partiremos de esas expresiones y comprobaremos que, con el algoritmo de síntesis, llegamos a los diagramas originales.

Ejemplo 1.

$$\begin{aligned}
 \alpha &= aa^*bb^* = (a)(a^*bb^*) \\
 D_a(\alpha) &= D_a(a)a^*bb^* + \delta(a) D_a(a^*bb^*) = \\
 &= \lambda a^*bb^* + \emptyset D_a(a^*bb^*) = a^*bb^* \\
 D_b(\alpha) &= D_b(a)a^*bb^* + \delta(a) D_b(a^*bb^*) = \emptyset a^*bb^* + \emptyset D_b(a^*bb^*) = \emptyset \\
 D_{aa}(\alpha) &= D_a(a^*)bb^* + \delta(a^*) D_a(bb^*) = \\
 &= D_a(a)a^*bb^* + \lambda \emptyset = \lambda a^*bb^* = a^*bb^* = D_a(\alpha)
 \end{aligned}$$

Al repetirse la derivada, ya no seguimos derivando respecto de a . Tampoco es necesario derivar $D_b(\alpha)$, puesto que, al ser \emptyset , cualquier derivada posterior será \emptyset . Nos queda pues por calcular $D_{ab}(\alpha)$:

$$\begin{aligned}
 D_{ab}(\alpha) &= D_b(a^*bb^*) = D_b(a^*)bb^* + \delta(a^*) D_b(bb^*) = \\
 &= D_b(a)a^*bb^* + \lambda [D_b(b)b^* + \delta(b) D_b(b^*)] = \\
 &= \emptyset a^*bb^* + \lambda b^* + \emptyset \lambda b^* = b^*
 \end{aligned}$$

Calcularemos ahora las derivadas terceras a partir de $D_{ab}(\alpha)$ (puesto que ésta es la única derivada segunda que no es igual a ninguna anterior).

$$\begin{aligned}
 D_{aba}(\alpha) &= D_a(b^*) = D_a(b)b^* = \emptyset b^* = \emptyset \\
 D_{abb}(\alpha) &= D_b(b^*) = D_b(b)b^* = b^* = D_{ab}(\alpha)
 \end{aligned}$$

Al salir una \emptyset y la otra repetida ya no es preciso que sigamos derivando.

Según el algoritmo, el conjunto de estados será:

$$Q = \{\alpha, D_a(\alpha), D_b(\alpha), D_{ab}(\alpha)\}$$

De las derivadas calculadas, la única que contiene la cadena vacía es $D_{ab}(\alpha) = b^*$ por lo que

$$F = \{D_{ab}(\alpha)\}$$

Y la función de transición será

$$f(a, \alpha) = D_a(\alpha); f(b, \alpha) = D_b(\alpha)$$

$$f(a, D_a(\alpha)) = D_{aa}(\alpha) = D_a(\alpha)$$

$$f(b, D_a(\alpha)) = D_{ab}(\alpha)$$

$$f(a, D_b(\alpha)) = D_{ba}(\alpha) = D_b(\alpha)$$

$$f(b, D_b(\alpha)) = D_{bb}(\alpha) = D_b(\alpha)$$

$$f(a, D_{ab}(\alpha)) = D_{aba}(\alpha) = D_b(\alpha)$$

$$f(b, D_{ab}(\alpha)) = D_{abb}(\alpha) = D_{ab}(\alpha)$$

Podemos así dibujar el diagrama de la figura 4.5, que es el mismo de la figura 4.1, con $q_1 = \alpha$; $q_2 = D_a(\alpha)$; $q_3 = D_{ab}(\alpha)$; $q_4 = D_b(\alpha)$.

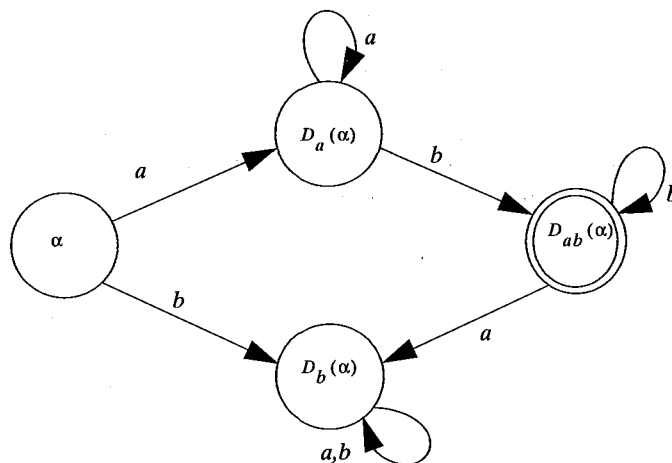


Figura 4.5.

Ejemplo 2.

La expresión regular es $\alpha = 1(01)^*$. Calculemos las derivadas hasta que aparezcan repetidas:

$$\begin{aligned}
D_0(\alpha) &= D_0(1)(01)^* + \delta(1)D_0(01)^* = \emptyset \\
D_1(\alpha) &= D_1(1)(01)^* + \delta(1)D_1(01)^* = (01)^* \\
D_{00}(\alpha) &= D_{01}(\alpha) = \emptyset \\
D_{10}(\alpha) &= D_0[(01)^*] = [D_0(01)](01)^* = 1(01)^* = \alpha \\
D_{11}(\alpha) &= D_1[(01)^*] = \emptyset
\end{aligned}$$

Por tanto,

$$\begin{aligned}
Q &= \{\alpha, D_0(\alpha), D_1(\alpha)\} \\
F &= \{D_1(\alpha)\}
\end{aligned}$$

La función de transición será:

$$\begin{aligned}
f(0, \alpha) &= D_0(\alpha) \\
f(1, \alpha) &= D_1(\alpha) \\
f(0, D_0(\alpha)) &= D_{00}(\alpha) = D_0(\alpha) \\
f(1, D_0(\alpha)) &= D_{01}(\alpha) = D_0(\alpha) \\
f(0, D_1(\alpha)) &= D_{10}(\alpha) = \alpha \\
f(1, D_1(\alpha)) &= D_{11}(\alpha) = D_0(\alpha)
\end{aligned}$$

Llegamos así a un diagrama de Moore como el de la figura 4.6 que, aparentemente, corresponde a un reconocedor diferente del original (figura 4.2). No hay ningún error. Lo que ocurre es que este algoritmo que estamos aplicando nos da el reconocedor en forma mínima, y el de la figura 4.2 no lo está.

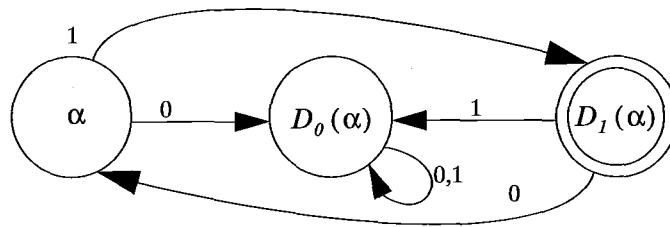


Figura 4.6.

Invitamos al lector a aplicar el algoritmo de minimización explicado en el capítulo 2 al AF de la figura 4.2 para ver que se llega al mismo AF de la figura 4.6.

Ejemplo 3.

$$\alpha = a + bc$$

Derivadas

$$D_a(\alpha) = D_a(a) + D_a(bc) = \lambda + \emptyset = \lambda$$

$$D_b(\alpha) = D_b(a) + D_b(bc) = \emptyset + c = c$$

$$D_c(\alpha) = D_c(a) + D_c(bc) = \emptyset$$

$$D_{aa}(\alpha) = D_{ab}(\alpha) = D_{ac}(\alpha) = \emptyset = D_c(\alpha)$$

$$D_{ba}(\alpha) = D_a(c) = \emptyset = D_c(\alpha)$$

$$D_{bb}(\alpha) = D_b(c) = \emptyset = D_c(\alpha)$$

$$D_{bc}(\alpha) = D_c(c) = \lambda = D_a(\alpha)$$

$$D_{ca}(\alpha) = D_{cb}(\alpha) = D_{cc}(\alpha) = \emptyset = D_c(\alpha)$$

$$Q = \{\alpha, D_a(\alpha), D_b(\alpha), D_c(\alpha)\}$$

$$F = \{D_a(\alpha)\}$$

Función de transición

$$f(a, \alpha) = D_a(\alpha); f(b, \alpha) = D_b(\alpha); f(c, \alpha) = D_c(\alpha)$$

$$f(a, D_a(\alpha)) = D_{aa}(\alpha) = D_c(\alpha)$$

$$f(b, D_a(\alpha)) = D_{ab}(\alpha) = D_c(\alpha)$$

$$f(c, D_a(\alpha)) = D_{ac}(\alpha) = D_c(\alpha)$$

$$f(a, D_b(\alpha)) = D_{ba}(\alpha) = D_c(\alpha)$$

$$f(b, D_b(\alpha)) = D_{bb}(\alpha) = D_c(\alpha)$$

$$f(c, D_b(\alpha)) = D_{bc}(\alpha) = D_a(\alpha)$$

$$f(a, D_c(\alpha)) = D_{ca}(\alpha) = \emptyset$$

$$f(b, D_c(\alpha)) = D_{cb}(\alpha) = \emptyset$$

$$f(c, D_c(\alpha)) = D_{cc}(\alpha) = \emptyset$$

Con esto, resulta el diagrama de la figura 4.7, que es distinto del original de la figura 4.3. La explicación es la misma del ejemplo anterior: el de la figura 4.3 no está en forma mínima (q_2 y q_4 son equivalentes).

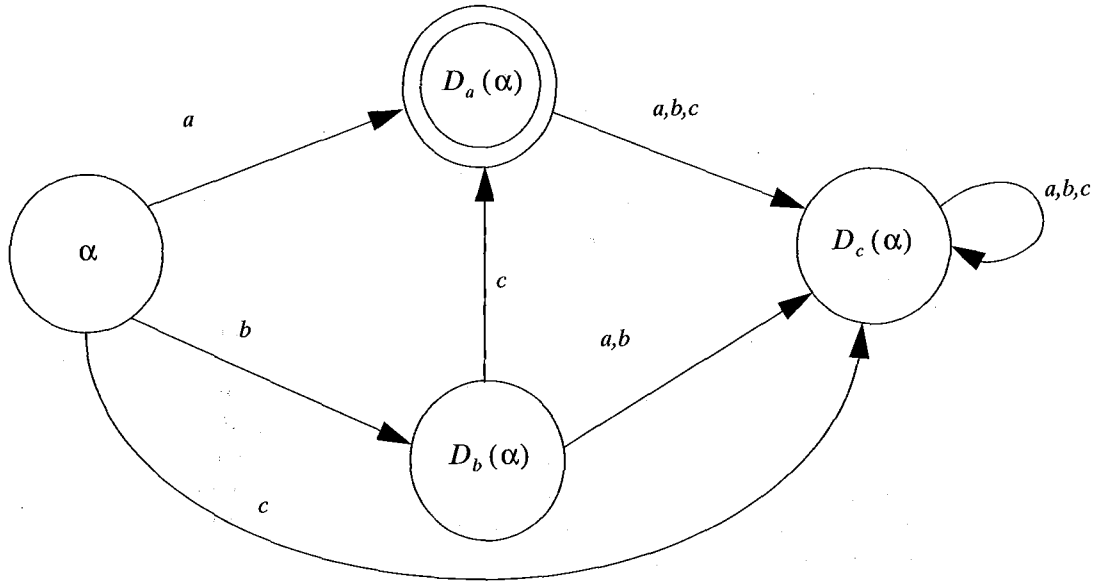


Figura 4.7.

Ejemplo 4.

$$\alpha = (1 + 2)^*3$$

Derivadas:

$$\begin{aligned} D_1(\alpha) &= D_1[(1 + 2)^*]3 + \delta[(1 + 2)^*]D_1(3) = \\ &= [D_1(1 + 2)](1 + 2)^*3 + \lambda \emptyset = (1 + 2)^*3 = \alpha \end{aligned}$$

$$D_2(\alpha) = \dots = (1 + 2)^*3 = \alpha$$

$$D_3(\alpha) = D_3[(1 + 2)^*]3 + \delta[(1 + 2)^*]D_3(3) = \emptyset + \lambda \lambda = \lambda$$

$$D_{31}(\alpha) = D_{32}(\alpha) = D_{33}(\alpha) = \emptyset$$

$$Q = \{\alpha, D_3(\alpha), D_{31}(\alpha)\}$$

$$F = \{D_3(\alpha)\}$$

Calculando los valores de la función de transición se llega a un diagrama idéntico al de la figura 4.4, con $q_1 = \alpha$, $q_2 = D_3(\alpha)$, $q_3 = D_{31}(\alpha)$.

5. Resumen

Orientándonos ya hacia las relaciones entre lenguajes y autómatas, hemos definido un reconocedor finito como un tipo particular de autómata finito en el que existe un estado inicial fijo y el alfabeto de salida consta sólo de dos símbolos, correspondientes a la aceptación o no aceptación de la cadena de entrada. Hemos demostrado la condición general que debe cumplir un lenguaje para que todas sus cadenas sean aceptadas por un reconocedor finito: que la relación de congruencia derecha inducida por el lenguaje tenga índice finito (ese índice es, precisamente, igual al número de estados del reconocedor).

Se han definido los conjuntos regulares, que, según los teoremas de análisis y síntesis, son justamente los lenguajes que pueden ser aceptados por un reconocedor finito. Las expresiones regulares constituyen un metalenguaje para describir de una manera cómoda a los conjuntos regulares. El algoritmo de análisis permite, dado un reconocedor finito, deducir la expresión regular del lenguaje que acepta ese reconocedor. A la inversa, hemos visto un algoritmo de síntesis mediante el cual se llega al diagrama de Moore de un reconocedor finito minimizado correspondiente a una expresión regular dada.

6. Notas histórica y bibliográfica

Los conceptos de conjunto regular y expresión regular, así como los teoremas de análisis y síntesis, se deben a Kleene (1956). La demostración del teorema de análisis que hemos seguido es la de McNaughton y Yamada (1960).

Para el teorema de síntesis existen varias demostraciones. Quizá la más utilizada sea la debida a Rabin y Scott (1959), que precisa de la introducción de un tipo especial de autómata que no tiene interpretación física: el autómata no determinista (o "posibilístico", como prefiere llamarlo Arbib (1969), ya que no interviene la idea de probabilidad), que veremos en el tema "Lenguajes" (capítulo 4, apartado 5). Una generalización de este autómata es el llamado sistema de transición (Ott y Feinstein, 1961), que permite llegar a un algoritmo bastante cómodo para la síntesis. Sin embargo, nos ha parecido que el algoritmo de síntesis más manejable es el basado en el trabajo de Brozozowski (1962, 1965), que introdujo el concepto de derivada de una expresión regular.

Las anteriores referencias tienen un interés más bien histórico. Para ampliar este capítulo es preferible dirigirse a libros de carácter general, como los ya citados en el capítulo 2, o el de Hopcroft y Ullman (1979). Los cuatro ejemplos utilizados reiteradamente están adaptados de Scala y Minguet (1974).

7. Ejercicios

7.1. Dada la tabla de transición

	<i>a</i>	<i>b</i>	<i>c</i>
q_1	q_2	q_6	q_1
q_2	q_3	q_5	q_1
q_3	q_3	q_4	q_1
q_4	q_7	q_7	q_1
q_5	q_6	q_5	q_1
q_6	q_7	q_6	q_1
q_7	q_7	q_7	q_1

correspondiente a un reconocedor finito, hallar la expresión regular correspondiente al lenguaje aceptado por ese reconocedor.

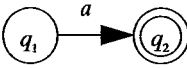
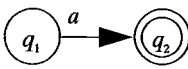
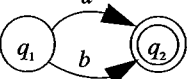
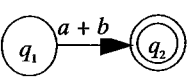

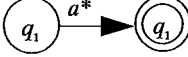
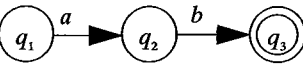
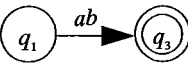
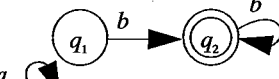
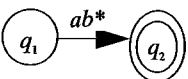
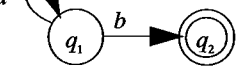
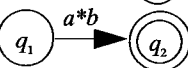
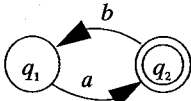
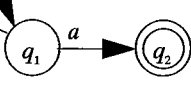
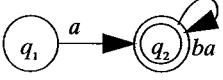
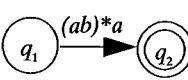
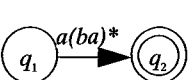
- 7.2. Aplicar el algoritmo de síntesis a la expresión regular encontrada en el ejercicio anterior.
- 7.3. Aplicar el algoritmo de análisis al reconocedor de la cadena 010 (capítulo 2, apartado 4.5.2) y al reconocedor de 321 (capítulo 2, ejercicio 9). En esos ejemplos se utilizó "reconocedor" con un sentido distinto del definido en este capítulo. ¿Puedes indicar cuáles son las diferencias?
- 7.4. Dar una expresión regular correspondiente al detector de paridad par y otra al detector de paridad impar, y aplicar el algoritmo de síntesis para obtener los correspondientes reconocedores.
- 7.5. Diseñar un circuito secuencial que dé una salida "1" cuando la cadena de entrada tenga un número de "unos" congruente a 0 (mód. 2). (La expresión regular es $\alpha = 0^*(10^*10^*)^*$).
- 7.6. Considérese un reconocedor finito definido por:

$$E = \{a, b, c, d\}; Q = \{q_1, q_2, q_3, q_4\}; F = \{q_4\}$$

$q \backslash e$	a	b	c	d
q_1	q_2	q_3	q_4	q_4
q_2	q_3	q_2	q_4	q_4
q_3	q_3	q_3	q_3	q_3
q_4	q_3	q_3	q_4	q_4

Hallar una expresión regular de las cadenas aceptadas por el reconocedor, y diseñar un circuito secuencial para la realización del reconocedor con biestables *JK*.

Los dos ejercicios siguientes, tomados de Booth (1967), desarrollan una técnica gráfica para obtener la expresión regular directamente del diagrama de Moore.

	Expresión regular	Diagramas	Diagrama reducido
1)	a		
2)	$a + b$		
3)	a^*		
4)	ab		
5)	ab^*		
6)	a^*b		
7)	$(ab)^*a = a(ba)^*$	<div style="display: flex; align-items: center;"> { <div style="display: flex; flex-direction: column; gap: 10px;">    </div> </div>	<div style="display: flex; align-items: center;"> { <div style="display: flex; flex-direction: column; gap: 10px;">   </div> </div>

7.7. Las expresiones regulares básicas que ilustra el diagrama de la página anterior describen a los conjuntos regulares de los reconocedores correspondientes. (El estado inicial es siempre q_1):

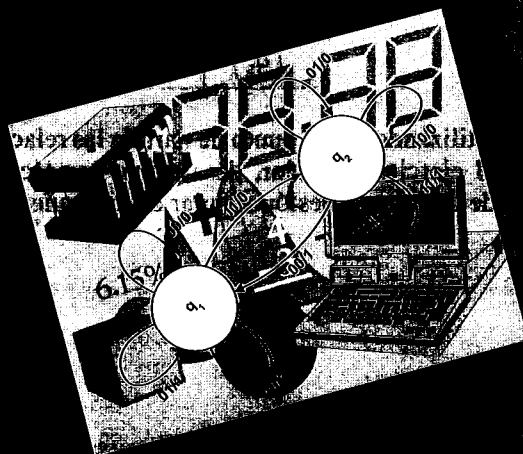
- a) Comprobar que las expresiones regulares corresponden efectivamente a los diagramas.
- b) Utilizar esas relaciones para obtener la expresión regular del reconocedor dado por la tabla:

	a	b
q_1	q_1	q_2
q_2	q_4	q_5
q_3	q_5	q_1
q_4	q_3	q_2
q_5	q_4	q_5

- 7.8.** Utilizando como punto de partida las relaciones básicas establecidas en el ejercicio anterior, desarrollar una técnica gráfica de análisis para deducir la expresión regular de cualquier reconocedor finito.

Fundamentos de informática

5



5

Otros autómatas

1. Introducción

El autómata finito estudiado en los anteriores capítulos es un modelo aplicable a cualquier sistema dinámico, discreto y con memoria (siempre que la memoria sea finita). Ahora bien, para determinados sistemas, la descripción que se obtendría con tal modelo sería inútil, por demasiado compleja (número de estados y de transiciones desorbitado) o porque los estados engloban de forma poco natural un conjunto de condiciones internas del sistema. Tal ocurre, por ejemplo, cuando el sistema está compuesto por subsistemas con funcionamiento asíncrono y concurrente. Para modelar estos casos se han propuesto diversas generalizaciones del modelo. Una de las más conocidas y más aplicadas en informática es la llamada "red de Petri". La red de Petri puede, además, como veremos, modelar sistemas con infinitos estados, por lo que se aproxima, en cuanto a potencia de descripción, a la máquina de Turing. Otra posibilidad para la modelación de sistemas complejos es construir redes de autómatas interconectados, llamados "autómatas celulares".

Por otra parte, los modelos que hemos considerado hasta ahora son completamente deterministas. Es decir, dado un estado y un símbolo de entrada, el estado siguiente y el símbolo de salida vienen determinados por las funciones f y g , respectivamente. Pero se dan situaciones cuya complejidad funcional y la imposibilidad de analizar todos los factores que intervienen hacen que sea preferible

modelarlas a partir de la idea de probabilidad. Tal ocurre, por ejemplo, cuando la fiabilidad de los componentes es pequeña, o cuando se quiere modelar órganos o procesos de los sistemas vivos. Así, si observamos que, estando en el estado q_1 y recibiendo entrada e_1 , el sistema pasa al estado q_2 un 80% de las veces y al q_3 un 20%, diremos que $f(e_1, q_1) = q_2$ con probabilidad 0,8 y $f(e_1, q_1) = q_3$ con probabilidad 0,2. Esta es la idea básica del autómata probabilista o estocástico.

Si en lugar de "probabilidad" utilizamos "borrosidad" tendremos un autómata borroso. Uno de los campos de aplicación más interesante de los autómatas estocásticos y de los borrosos está en los sistemas de aprendizaje: el autómata, en interacción con su entorno, puede ir modificando su estructura para adaptarse a tal entorno. En este capítulo trataremos de todo esto (redes de Petri, autómatas estocásticos y borrosos, autómatas celulares y aprendizaje).

2. Redes de Petri

2.1. Estructura estática

Definición 2.1.1. Una *red de Petri* (RdP) es una cuádrupla

$$\langle L, T, \alpha, \beta \rangle$$

donde:

L : es un conjunto finito, no vacío, de *lugares*.

T : es un conjunto finito, no vacío, y disjunto de L , de *transiciones*.

$\alpha: T \rightarrow P(L)$ es la *función de incidencia anterior*, o *función de entrada*, que define, para cada transición, sus lugares de entrada.

$\beta: T \rightarrow P(L)$ es la *función de incidencia posterior*, o *función de salida*, que define, para cada transición, sus lugares de salida.

Por ejemplo:

$$L = \{l_1, l_2, l_3, l_4, l_5\}$$

$$T = \{t_1, t_2, t_3, t_4\}$$

$$\alpha(t_1) = \{l_1, l_5\}; \beta(t_1) = \{l_2\}$$

$$\alpha(t_2) = \{l_2\}; \beta(t_2) = \{l_1, l_5\}$$

$$\alpha(t_3) = \{l_3, l_5\}; \beta(t_3) = \{l_4\}$$

$$\alpha(t_4) = \{l_4\}; \beta(t_4) = \{l_3, l_5\}$$

Una RdP puede representarse gráficamente como un grafo bipartido orientado, en el que hay dos tipos de nodos: círculos (para los lugares) y segmentos (para las transiciones), y los arcos representan a las funciones α y β . Así, el grafo del ejemplo anterior sería el de figura 5.1.

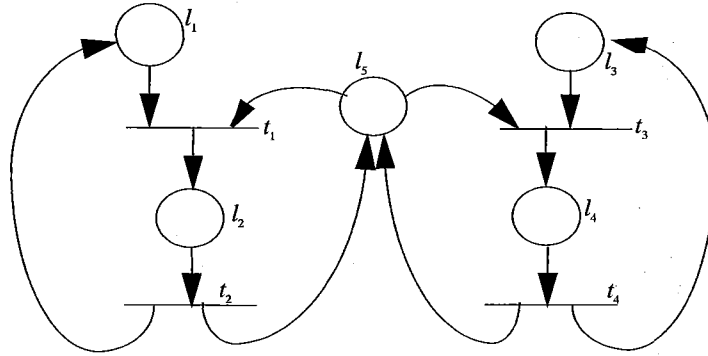


Figura 5.1.

Consideraremos solamente las llamadas "RdP puras", en las que no puede haber lugares que correspondan a una entrada y salida de la misma transición. Es decir,

$$(\forall t_i) \quad (\alpha(t_i) \cap \beta(t_i) = \emptyset)$$

En este caso, la RdP puede representarse también mediante una *matriz de incidencia* cuyas filas corresponden a los lugares y cuyas columnas corresponden a las transiciones y cuyo elemento (i, j) es:

- 0 si l_i no es entrada ni salida de t_j
- 1 si l_i es salida de t_j
- 1 si l_i es entrada de t_j

La matriz de incidencia del ejemplo anterior sería:

$$\begin{matrix} & t_1 & t_2 & t_3 & t_4 \\ \begin{matrix} l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \end{matrix} & \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \end{matrix}$$

El concepto de RdP tal como lo acabamos de definir está incompleto, porque no contempla ningún comportamiento dinámico. En particular, se observará que no se ha definido el estado.

2.2. Comportamiento dinámico

Definición 2.2.1. Un *marcado* de una RdP es una asignación de *marcas* (números naturales) a los lugares:

$$\mu : L \rightarrow N$$

El número de marcas en l_i será $\mu(l_i)$. El número y posición de las marcas varía con el tiempo según las reglas que definiremos enseguida. En cada instante, tendremos un vector $\mu = [\mu(l_1), \mu(l_2) \dots]^T$ que indica el número de marcas en cada lugar.

Definición 2.2.2. Una RdP marcada es una quintupla

$$\langle L, T, \alpha, \beta, \mu_0 \rangle$$

donde los cuatro primeros elementos son los de la Definición 2.1.1, y μ_0 es un *marcado inicial*.

En adelante consideraremos siempre RdP marcadas.

Gráficamente, el marcado se indica mediante puntos en los círculos que representan a los lugares. Así, si $\mu_0 = [1 \ 0 \ 1 \ 0 \ 1]^T$, el grafo de la RdP marcada correspondiente al ejemplo anterior será el de la figura 5.2.

Definición 2.2.3. Una transición t_i está *permitida* por el marcado si todos sus lugares de entrada tienen al menos una marca, es decir, si

$$(\forall l_j \in \alpha(t_i)) \quad (\mu(l_j) > 0)$$

En la figura 5.2, t_1 y t_3 están permitidas; t_2 y t_4 , no.

Definición 2.2.4. El *disparo* de una transición permitida consiste en quitar una marca de cada uno de sus lugares de entrada y añadir una marca a cada uno de sus lugares de salida.

Por ejemplo, el disparo de t_1 en la figura 5.2 conduce a un nuevo marcado: desaparecen las marcas de l_1 y l_5 y aparece una en t_2 ; el nuevo marcado es $\mu_1 = [0 \ 1 \ 1 \ 0 \ 0]^T$. Obsérvese que el hecho de disparar t_1 hace que t_3 deje de estar permitida.

Definición 2.2.5. El *estado* de una RdP en un instante viene dado por su marcado μ_i en ese instante. El conjunto de estados, M , será, pues, el conjunto de todos los marcados posibles.

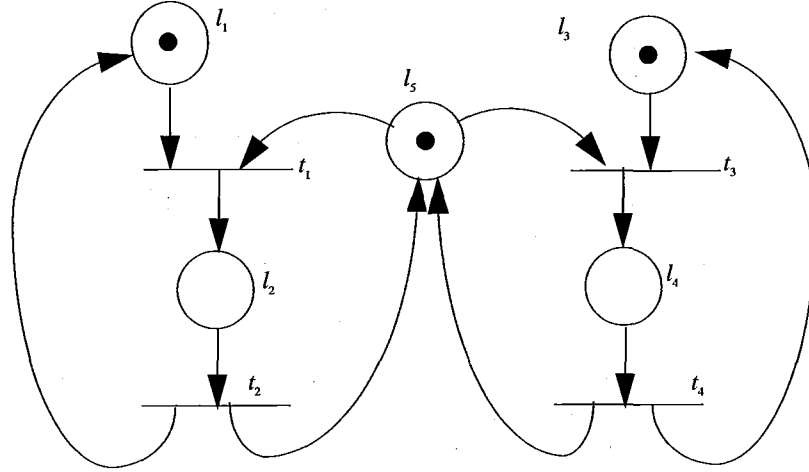


Figura 5.2.

Definición 2.2.6. La *función de transición*, f , de una RdP es una función

$$f: M \times T \rightarrow M$$

tal que aplicada a un marcado μ_i y a una transición t_j da el marcado que resulta de disparar t_j . Es una función parcial, porque si t_j no está permitida por μ_i entonces $f(\mu_i, t_j)$ no está definida.

Definición 2.2.7. Se llama *ejecución* de una RdP a la secuencia de acontecimientos que resulta de disparar transiciones a partir de un marcado inicial. Esta secuencia de acontecimientos se manifiesta en dos secuencias de objetos: una secuencia de marcados (μ_0, μ_1, \dots) , y una secuencia de transiciones (t_0, t_1, \dots) , tales que $f(\mu_i, t_i) = \mu_{i+1}$. En el ejemplo de la figura 5.2 podemos tener esta ejecución:

$$\mu_0 = [1 \ 0 \ 1 \ 0 \ 1]^T;$$

$$\mu_1 = [0 \ 1 \ 1 \ 0 \ 0]^T;$$

$$\mu_2 = \mu_0;$$

$$\mu_3 = [1 \ 0 \ 0 \ 1 \ 0]^T;$$

$$\mu_4 = \mu_0;$$

$$\mu_5 = \mu_1;$$

.....

$$(t_1, t_2, t_3, t_4, t_1, \dots)$$

Obsérvese que la ejecución no es única: con el marcado μ_0 pueden dispararse indistintamente t_1 ó t_3 ; cuál de ellas lo hace depende de acontecimientos externos que no figuran en el modelo. Las RdP son, pues, modelos no deterministas, como los autómatas que veremos en el tema "Lenguajes".

Teorema 2.2.8. La función de transición de una RdP puede escribirse explícitamente en función de la matriz de incidencia, I , y de un vector, u_i , cuyas componentes (tantas como transiciones tenga la RdP) son todas nulas, salvo la que corresponde a la transición disparada en el instante i , que vale 1, y ello mediante la llamada *ecuación de estado* de la RdP:

$$\mu_{i+1} = \mu_i + I \cdot u_i$$

Para la demostración, que omitimos, basta con ver que todas las componentes de μ_i satisfacen la ecuación, teniendo en cuenta la definición de I (apartado 2.1).

En nuestro ejemplo, si, partiendo de μ_0 , se dispara t_1 , tendremos:

$$\mu_1 = \mu_0 + I \cdot u_0 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Y si luego se dispara t_2 :

$$\mu_2 = \mu_1 + I \cdot u_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} =$$

$$= \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

El marcado final resultante de una secuencia de disparos a partir de un marcado inicial, μ_0 , puede obtenerse aplicando sucesivamente la ecuación de estado:

$$\begin{aligned} \mu_i &= \mu_{i-1} + I \cdot u_i = \mu_{i-2} + I \cdot (u_{i-1} + u_i) = \\ &= \mu_{i-3} + I \cdot (u_{i-2} + u_{i-1} + u_i) = \dots \\ &= \mu_0 + I \cdot \sum_{j=1}^i u_j = \mu_0 + I \cdot \bar{s} \end{aligned}$$

donde \bar{s} es el *vector característico* asociado a la secuencia $s = (t_1, t_2, \dots)$: su j -ésima componente es igual al número de ocurrencias de t_j en S . En nuestro ejemplo, si $s = (t_1, t_2, t_3, t_4, t_1)$, $\bar{s} = [2 \ 1 \ 1 \ 1]^T$,

$$\begin{aligned} \mu_s &= \mu_0 + I \cdot \bar{s} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \\ &= \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{aligned}$$

2.3. Autómata finito equivalente a una red de Petri con espacio de estados finito

Tal como se ha definido el estado de una RdP, el número de estados posibles puede ser infinito, porque el número de marcas en cada lugar no está, en general,

limitado. Por ejemplo, en la RdP de la figura 5.3, la secuencia $t_1, t_2, t_1, t_2, \dots$ hace crecer indefinidamente el número de marcas en l_3 .

Pero en ciertas RdP el número de marcados posibles a partir de uno inicial es finito. Así ocurre en el ejemplo de la figura 5.2. Esa RdP es, además, *binaria*: a partir de un marcado que no asigne a cada lugar más de una marca, todos los marcados sucesivos son tales que cualquier lugar tiene una marca o ninguna. En este caso, como hay cinco lugares, el número de marcados binarios posibles es $2^5 = 32$. Sin embargo, no todos ellos son alcanzables a partir de uno inicial dado. Por ejemplo, si μ_0 es el de la figura 5.2, es decir,

$$\mu_0 = [1 \ 0 \ 1 \ 0 \ 1]^T,$$

puede comprobarse que sólo son alcanzables otros dos marcados:

$$\mu_1 = [0 \ 1 \ 1 \ 0 \ 0]^T$$

$$\mu_2 = [1 \ 0 \ 0 \ 1 \ 0]^T$$

El AF equivalente a esta RdP sería el de la figura 5.4.

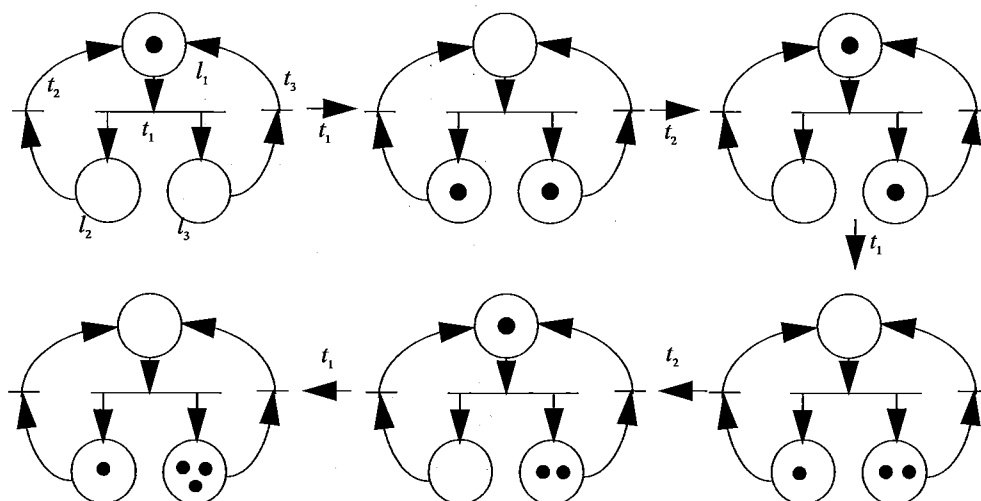


Figura 5.3.

2.4. Ejemplos de aplicación

2.4.1. Redes de Petri interpretadas

Para aplicar el concepto matemático de RdP a la modelación de sistemas hay que establecer una *interpretación* o convenio por el que se asocian las entradas

del sistema a condiciones necesarias para que ciertas transiciones se disparen y las salidas al disparo de otras transiciones o a las marcas de determinados lugares.

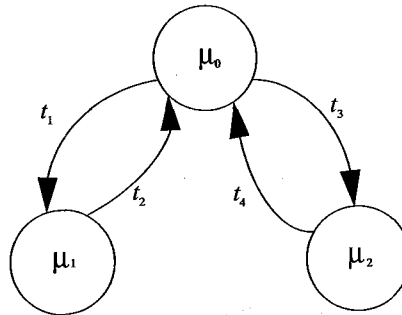


Figura 5.4.

Como ya hemos dicho en la introducción, las RdP son una herramienta de modelación útil para sistemas en los que se dan actividades asíncronas y concurrentes. Esto abarca campos de aplicación muy diversos. Aquí nos limitaremos a ilustrar con dos ejemplos su utilidad para abordar ciertos problemas que aparecen en el diseño de programas.

2.4.2. Problemas de exclusión mutua: lectores y redactores

Un problema que se presenta con frecuencia en los sistemas multiprogramados es el de evitar que dos procesos (programas en ejecución) puedan acceder simultáneamente a un recurso común no compartible: un periférico, un fichero, una tabla de datos en memoria, etc. Los segmentos de los programas que realizan ese acceso se llaman *secciones críticas*, y hay que dotar al sistema de mecanismos que aseguren la exclusión mutua de los procesos en la ejecución de tales secciones críticas.

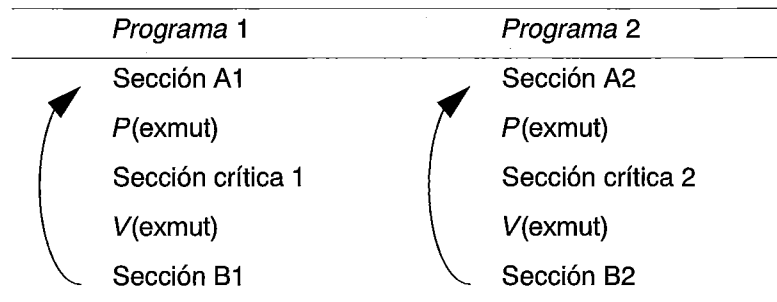
Uno de los mecanismos más utilizados es el del *semáforo* acompañado de las operaciones "P" (o "wait") y "V" (o "signal"). Un semáforo es una variable entera no negativa, S , que, aparte de su inicialización, sólo puede modificarse mediante las operaciones "P" y "V":

$P(S)$: si $S > 0$, disminuye S en una unidad

$V(S)$: incrementa S en una unidad

Si un proceso ejecuta "P" sobre un semáforo con $S = 0$, entonces debe esperar hasta que otro proceso ejecute "V" sobre el mismo semáforo. Por otra parte, son operaciones indivisibles: mientras un proceso no termina de ejecutar "P" o "V" otro no puede acceder al mismo semáforo.

Para asegurar la exclusión mutua de dos procesos a secciones críticas podemos utilizar un semáforo, "exmut", inicializado con el valor 1, del siguiente modo:



El primero de los procesos que ejecute P pondrá el semáforo a cero, y si el otro intenta entrar en su sección crítica antes de que el primero haya salido de la suya y ejecutado V , tendrá que esperar.

La modelación con una RdP puede hacerse considerando el semáforo como un lugar en el que el número de marcas es el valor del semáforo. Las operaciones P son transiciones de salida del lugar, y las V lo son de entrada. La RdP que modela el funcionamiento de los dos procesos anteriores es la de la figura 5.2, interpretada del siguiente modo:

l_5 es el semáforo "exmut".

t_1 es la " P " del Proceso 1; hay que añadir la condición de disparo: fin de la ejecución de la "Sección A1".

t_2 es la " V " del Proceso 1; la condición de disparo es el fin de la ejecución de la "Sección crítica 1".

l_1 es "entrada en la Sección B1".

l_2 es "entrada en la Sección crítica 1".

l_3 , t_3 , l_4 y l_5 son las análogas del Proceso 2.

Obsérvese que los lugares l_2 y l_4 quedan mutuamente excluidos.

Como un ejemplo más concreto del problema de la exclusión mutua, consideremos uno clásico: el de los lectores y redactores, procesos concurrentes que acceden a unos datos comunes con las siguientes restricciones:

- Los lectores pueden acceder a los datos simultáneamente.
- Por el contrario, los redactores, como modifican los datos, deben trabajar en exclusión mutua entre sí y excluyendo también a los lectores. Es decir, mientras un redactor está ejecutando la sección crítica (acceso a los datos) ningún otro proceso puede entrar en su sección crítica.
- Los procesos pueden estar en alguno de estos tres estados:
A (activo: ejecutando su sección crítica);

E (esperando entrar en su sección crítica);

R (reposo: no necesita los datos, aunque puede estar ejecutando otra parte del programa).

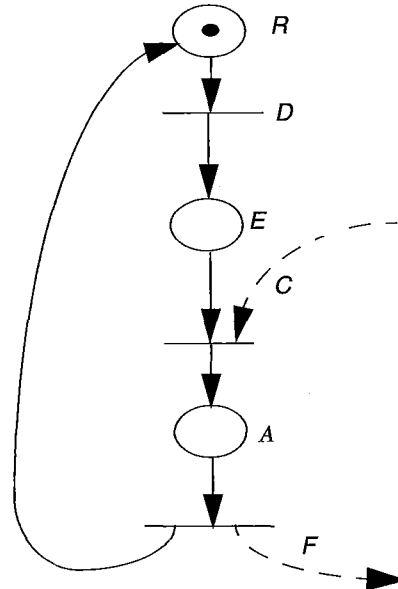


Figura 5.5.

Supongamos que hay dos lectores y dos redactores. Cada uno de los cuatro procesos puede modelarse como una RdP, según la figura 5.5. Los tres lugares corresponden a los estados definidos, y las transiciones se disparan cuando en el proceso aparecen determinados acontecimientos: *D* (demanda de acceso a los datos), *C* (comienzo de ejecución de la sección crítica, o acceso a los datos), *F* (fin de acceso a los datos).

Ahora bien, para poder cumplir las restricciones enunciadas será preciso interconectar las cuatro subredes: el acceso de un lector a los datos tienen que excluir acceso de los dos redactores, pero no del otro lector, y el acceso de un redactor tiene que excluir el acceso de los otros tres. Podemos utilizar dos semáforos, S_1 y S_2 , para excluir al lector 1 de los dos redactores y S_2 para el lector 2. La RdP global será la de la figura 5.6, en la que $l1$, $l2$, $r1$, $r2$ significan "lector 1", "lector 2", "redactor 1" y "redactor 2".

Obsérvese que la RdP en este ejemplo es binaria, y, por ello, el número de estados es finito. Por tanto, podríamos hallar un AF equivalente, como hacíamos en el apartado 2.3. Los estados de tal AF corresponderían a combinaciones de estados de cada uno de los procesos, haciendo más difícil la interpretación del modelo. Además, la RdP es modular: si se ha comprendido la idea de la figura 5.6, es muy fácil extenderla a casos con más lectores y/o redactores. Finalmente, el paso a la programación es inmediato: basta con poner, en cada proceso, operaciones *P* y *V* sobre S_1 o S_2 en las transiciones *C* y *F*.

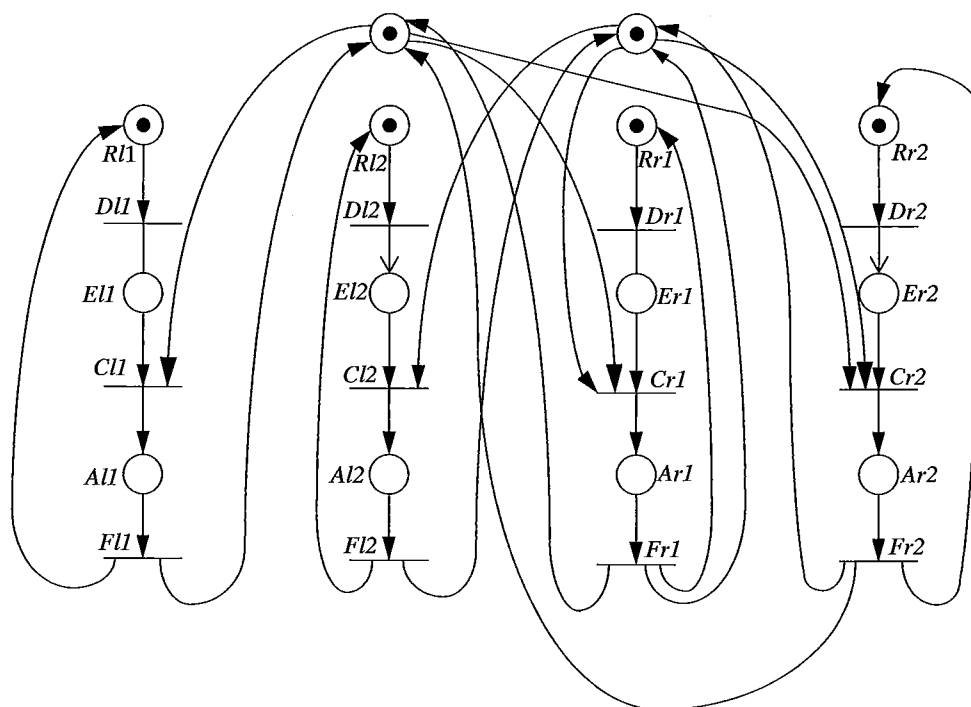


Figura 5.6.

2.4.3. Problemas de sincronización: productores y consumidores

En muchas aplicaciones, dos o más procesos pueden ejecutarse concurrentemente pero no de manera independiente: uno de los procesos no puede seguir a partir de un punto si el otro no ha realizado una cierta acción. Para concretar, consideremos otro problema clásico en informática: un proceso productor genera elementos que deposita en un *almacén* o zona de memoria ("buffer"), mientras que un proceso consumidor extrae elementos del mismo almacén. El almacén tiene una capacidad limitada, N . El productor repite continuamente un ciclo: "producir un elemento-meterlo en el almacén-producir un elemento..."; el consumidor, otro: "extraer un elemento-consumirlo-extraer...". En la figura 5.7, la parte de la izquierda corresponde al ciclo del productor, con

E_p = productor esperando (a que haya sitio en el almacén o a que el consumidor finalice su acceso);

C_m = comienzo de la operación de meter un elemento;

M = meter un elemento;

F_m = fin de la operación de meter (y comienzo de producir);

P = producir un elemento;

F_p = fin de la producción,

y la de la derecha, al ciclo del consumidor, con

E_c = consumidor esperando (a que haya elementos, o a que el productor finalice su acceso);

C_s = comienzo de la operación de sacar;

S = sacar;

F_s = fin de sacar (y comienzo de consumir);

C = consumir un elemento;

F_c = fin de consumir.

Como indican las líneas de puntos, la RdP está incompleta. Es preciso añadir los mecanismos de sincronización entre ambos procesos:

- a) C_m no puede dispararse si no hay sitio disponible;
- b) C_s no puede dispararse si no hay elementos disponibles;
- c) los procesos no pueden acceder simultáneamente al almacén (ello podría provocar, en la realización del programa, problemas de actualización de punteros).

Para ello, añadiremos tres semáforos, modelados en la RdP como tres lugares:

- a) S_s , que se inicializará con el valor N (capacidad del almacén), y cuyo valor (número de marcas) indicará el número de sitios disponibles. F_s lo incrementará en una unidad, y C_m lo decrementará.
- b) S_e , que se inicializará con valor 0, y cuyo valor indicará el número de elementos en el almacén. C_s lo decrementará, y F_m lo incrementará.
- c) S_m , semáforo binario con el valor inicial 1, que servirá para la exclusión mutua de ambos procesos en el acceso al almacén.

La RdP completa es la de la figura 5.8, en la que se ha puesto un marcado inicial que supone que la capacidad del almacén es de cuatro elementos.

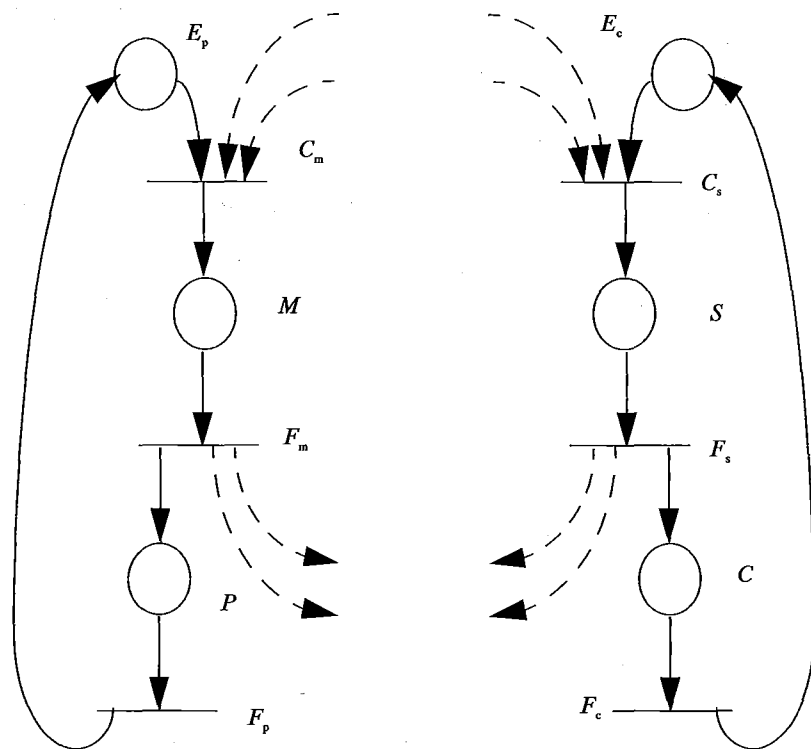


Figura 5.7.

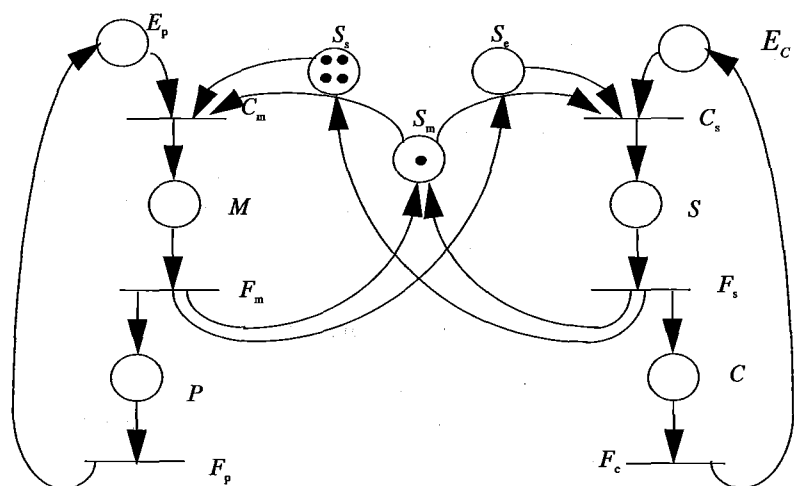
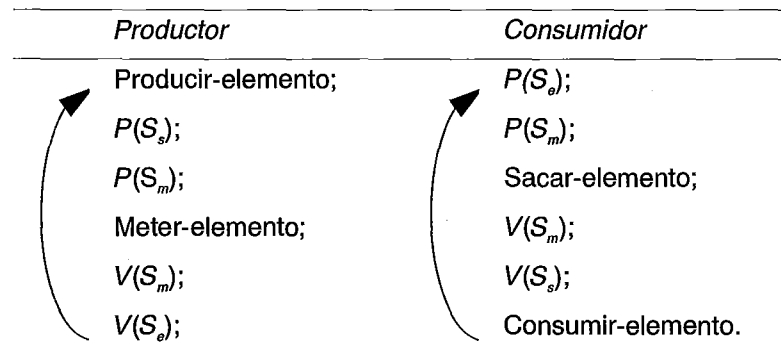


Figura 5.8.

Obsérvese también aquí que el número de estados es finito, pero que el AF equivalente, que podemos encontrar, modela de manera menos natural los fenómenos. Asimismo, que es fácil extender la RdP para que considere más productores y/o consumidores.

Un ejemplo típico de procesos que han de coordinarse de este modo se tiene en los sistemas operativos con los gestores ("drivers") de dispositivos de entrada y salida: un gestor de salida es un consumidor que, cuando tiene en el almacén alguna información para enviar al periférico de salida, se ejecuta; el productor es el programa que produce esas informaciones. Simétricamente, un gestor de entrada es un productor para los programas que consumen esas informaciones.

El esquema para la programación de un productor y un consumidor sería:



Obsérvese que S_m es el equivalente al semáforo "exmut" del ejemplo anterior, mientras que S_e y S_s aseguran la coordinación de los dos procesos.

3. Autómatas estocásticos

3.1. Definición

Un autómata estocástico o probabilista es una quintupla

$$AP = \langle E, S, Q, P, h \rangle,$$

donde:

E, S, Q : son, como en los autómatas deterministas, los alfabetos de entrada y salida y el conjunto de estados, que supondremos finito:

$$Q = \{q_1, q_2, \dots, q_n\}$$

$h: Q \rightarrow S$ es la función de salida (consideraremos sólo autómatas de tipo Moore), que se supone determinista.

$P: E \times Q \rightarrow [0,1]^n$ es la *función de probabilidades de transición*:

$$P(e, q) = (p_1(e, q), p_2(e, q), \dots, p_n(e, q)),$$

donde $0 \leq p_i(e, q) \leq 1$ es la probabilidad de que, estando en el estado q y recibiendo la entrada e , se pase al estado q_i . Deberá cumplirse que:

$$\sum_{i=1}^n p_i(e, q) = 1$$

Si consideramos todos los estados en que puede encontrarse el autómata cuando recibe una entrada e , podemos definir una *matriz de probabilidades de transición*, $M(e)$:

$$M(e) = \begin{bmatrix} p_1(e, q_1) & \dots & p_n(e, q_1) \\ \dots & & \dots \\ p_1(e, q_n) & \dots & p_n(e, q_n) \end{bmatrix}$$

de modo que el elemento $m_{ij} = p_j(e, q_i)$ de $M(e)$ es la probabilidad de pasar del estado q_i al q_j bajo la acción de la entrada e .

Si $x = e_1 e_2 \dots e_m \in E^*$, es fácil demostrar, por inducción sobre m , que $M(x) = M(e_1) \cdot M(e_2) \dots M(e_m)$.

Un AF *determinista es un caso particular del estocástico*. En el caso estocástico, $m_{ij} \in [0,1]$, y en el determinista $m_{ij} \in \{0,1\}$. En cualquier caso, la suma de los elementos de cada fila en todas las matrices M debe ser la unidad.

3.2. Ejemplo

Sea un autómata estocástico definido por:

$$E = S = \{0, 1\}$$

$$Q = \{q_1, q_2\}$$

$$h(q_1) = 0; h(q_2) = 1$$

$$P(0, q_1) = (0,7; 0,3); P(1, q_1) = (0,4; 0,6)$$

$$P(0, q_2) = (0,6; 0,4); P(1, q_2) = (0,2; 0,8)$$

Podemos representar este autómata por un diagrama de Moore (figura 5.9), indicando sobre cada transición la probabilidad asociada.

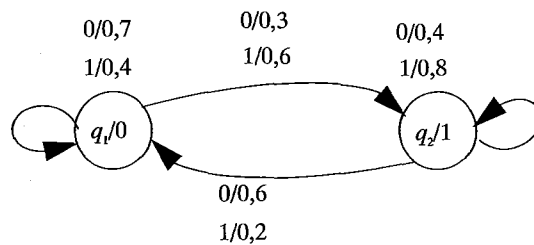


Figura 5.9.

Las matrices de probabilidades de transición correspondientes a los símbolos de entrada son:

$$M(0) = \begin{bmatrix} 0,7 & 0,3 \\ 0,6 & 0,4 \end{bmatrix}; M(1) = \begin{bmatrix} 0,4 & 0,6 \\ 0,2 & 0,8 \end{bmatrix}$$

Si por ejemplo estamos interesados en la respuesta a la cadena $x = 011$, tendremos:

$$M(011) = \begin{bmatrix} 0,7 & 0,3 \\ 0,6 & 0,4 \end{bmatrix} \cdot \begin{bmatrix} 0,4 & 0,6 \\ 0,2 & 0,8 \end{bmatrix}^2 = \begin{bmatrix} 0,268 & 0,732 \\ 0,264 & 0,736 \end{bmatrix}$$

Así, la probabilidad de pasar de q_1 a q_2 bajo la acción de $x = 011$ es 0,732. Al mismo resultado se llega si se consideran las cuatro posibilidades existentes para pasar de q_1 a q_2 con la cadena 011:

$$\xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{1} q_2 : \text{probabilidad} = 0,7 \times 0,4 \times 0,6 = 0,168$$

$$\xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_2 : \text{probabilidad} = 0,7 \times 0,6 \times 0,8 = 0,336$$

$$\xrightarrow{0} q_2 \xrightarrow{1} q_2 \xrightarrow{1} q_2 : \text{probabilidad} = 0,3 \times 0,8 \times 0,8 = 0,192$$

$$\xrightarrow{0} q_2 \quad \xrightarrow{1} q_1 \quad \xrightarrow{1} q_2 : \text{probabilidad} = 0,3 \times 0,2 \times 0,6 = 0,036$$

$$\text{probabilidad total} = 0,732$$

3.3. Reconocedores estocásticos

Siguiendo la misma línea del capítulo 4, podemos definir un reconocedor estocástico como

$$R_p = \langle E, Q, q_1, P, F \rangle$$

donde q_1 es el estado designado como inicial y $F \subset Q$ es el conjunto de estados finales. (Algunos autores introducen también la indeterminación de tomar un estado inicial u otro, y, en lugar de q_1 , incluyen un conjunto, $\Psi = \{\psi_1, \psi_2, \dots, \psi_n\}$, que representa las probabilidades asociadas a cada estado inicialmente).

El estudio de los lenguajes aceptados por reconocedores estocásticos es más complicado que en el caso determinista, ya que una misma cadena puede ser aceptada por el reconocedor en unas ocasiones y rechazada en otras. La probabilidad de que $x \in E^*$ sea aceptada, será:

$$p(x) = \sum_i p_i(q_1, x)$$

Así, en el ejemplo anterior, si q_1 es el estado inicial y q_2 el final, $p(011) = 0,732$.

Se define el lenguaje aceptado por un reconocedor estocástico haciéndolo depender de un parámetro designado mediante la letra λ , llamado *punto de corte* del lenguaje:

$$L_{RP}(\lambda) = \{x \in E^* \mid p(x) > \lambda\}$$

y se demuestra que tales lenguajes son una generalización de los conjuntos regulares.

4. Autómatas estocásticos de estructura variable

Un autómata estocástico o probabilista de estructura variable, APEV, es una séxtupla

$$\text{APEV} = \langle E, S, Q, P, h, A \rangle,$$

donde E, S, Q, P, h tienen el mismo significado anterior, y A es un algoritmo llamado *esquema de actualización o esquema de refuerzo* que genera P_{i+1} a partir de P_i, s , y e . Así en un APEV las probabilidades de transición no son siempre las mismas, sino que van evolucionando en función de su valor anterior, de la respuesta última y de la entrada. *Esta característica es la que confiere al APEV la capacidad de adaptarse y de aprender.*

5. Autómatas con aprendizaje

5.1. Una primera definición de "aprendizaje"

De momento, podemos definir el término aprendizaje como la *capacidad que tiene un sistema para cambiar su estructura adaptativamente, de modo que las respuestas que genera a los estímulos de su entorno van siendo progresivamente mejores.*

Así pues, para decidir si las respuestas son "mejores", un elemento externo al sistema (que, por tanto, forma parte del entorno) ha de evaluar esas respuestas. El sistema que aprende consta de dos subsistemas: uno contiene la parte adaptativa, es decir, los componentes que cambian (en los casos más sencillos y más estudiados, estos cambios son, simplemente, modificaciones de parámetros), y el otro, al que llamaremos *esquema de actualización o algoritmo de aprendizaje*, provoca los cambios de acuerdo con el resultado de la evaluación. La figura 5.10 muestra un esquema genérico del conjunto formado por el sistema que aprende y el entorno.

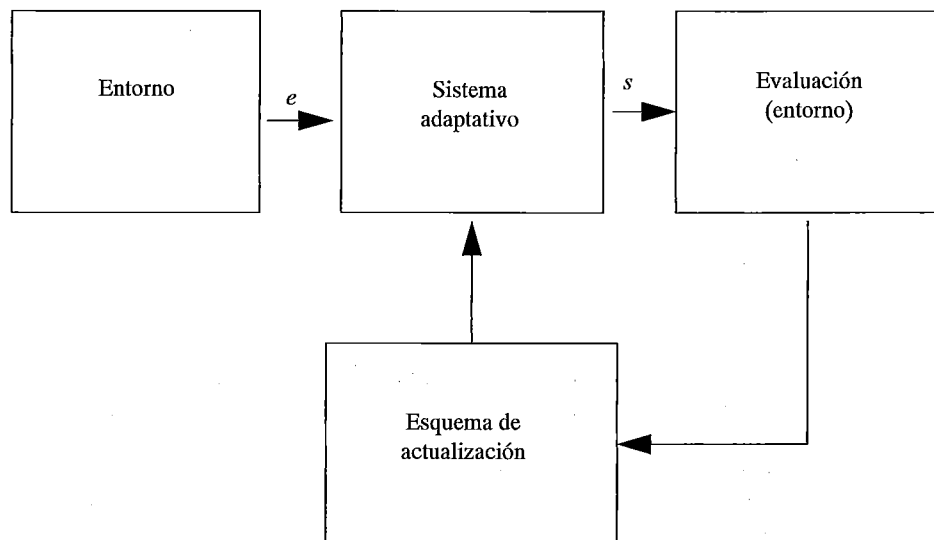


Figura 5.10.

En el apartado 8 veremos que esta definición se puede enmarcar en una concepción más amplia del aprendizaje. Veamos ahora cómo con autómatas sencillos pueden construirse sistemas cuyo comportamiento satisface a esta definición.

5.2. Aprendizaje en un APEV

Un APEV contiene los dos subsistemas mencionados: las probabilidades P son los parámetros que cambian (la parte adaptativa), y el algoritmo A es el esquema de actualización. Por tanto, interactuando con un entorno que evalúe sus salidas (figura 5.11) puede presentar un comportamiento que se ajuste a la definición dada de "aprendizaje". Podemos así definir un *autómata de aprendizaje probabilista* como un APEV que opera en un entorno y va actualizando sus probabilidades de transición para mejorar su comportamiento en algún sentido especificado.

En psicología, un autómata de aprendizaje puede ser un modelo del comportamiento de un organismo bajo estudio, donde el APEV será el modelo del organismo y el entorno estará representado por el experimentador. En una aplicación de ingeniería, tal como el control de un proceso, el controlador es el APEV, y el resto del sistema, con sus incertidumbres, constituye el entorno.

Las respuestas del entorno son frecuentemente binarias, es decir, $E = \{0, 1\}$, y a una de ellas se le llama "respuesta de premio" y a la otra "respuesta de castigo".

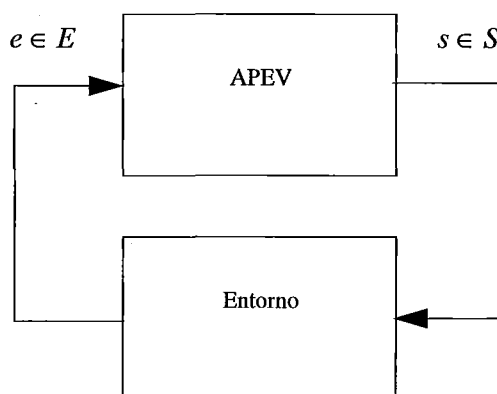


Figura 5.11.

La utilización de un autómata de aprendizaje sólo tiene interés cuando el comportamiento del entorno es desconocido, puesto que si no fuera así, un AF determinista nos resolvería el problema. Generalmente se supone que el entorno es aleatorio, y la probabilidad de que produzca salida "1" dependerá de su entrada, es decir, de la respuesta s del autómata. Así, si S tiene r elementos, tendremos r probabilidades $C_i (i = 1, 2, \dots, r)$ de que el entorno de salida "1". A estas probabilidades se les llama *probabilidades de castigo*.

Para juzgar el grado de aprendizaje se define un *castigo medio recibido por el autómata*: en un instante t , si el autómata produce la salida s_i con probabilidad $p_i(t)$, el castigo medio condicionado a $P(t)$ es:

$$M(t) = E[e(t) | P(t)] = \sum_{i=1}^r p_i(t) C_i$$

Si suponemos que en $t = 0$ el autómata escogerá una salida u otra con igual probabilidad, se tendrá

$$M(0) = \frac{\sum C_i}{r}$$

El uso del término "aprendizaje" sólo tiene sentido si $M(t)$ se va haciendo menor que $M(0)$. Según como sea esa tendencia se definen unos tipos u otros de autómatas, y se estudian los esquemas de actualización necesarios para conseguirlos.

6. Autómatas borrosos

Podemos definir un autómata finito borroso como

$$AB = \langle E, S, Q, f, h \rangle$$

donde E, S, Q y $h : Q \rightarrow S$ tienen el mismo significado que en el AF ordinario, y f es una *función de transición borrosa*:

$$f: E \times Q \times Q \rightarrow M$$

(en adelante tomaremos $M = \{x \mid 0 \leq x \leq 1\}$). Esta f se interpreta así:

si $f(e, q_i, q_j) = 0$, seguro que, con entrada e , *no hay* transición de q_i a q_j ;

si $f(e, q_i, q_j) = 1$, seguro que, con entrada e , *hay* transición de q_i a q_j ;

si $f(e, q_i, q_j) = \alpha$, α es el *grado de confianza* de que, para entrada e , haya transición de q_i a q_j .

(Obsérvese que también podríamos haber definido, de forma totalmente equivalente,

$$F: E \times Q \rightarrow M$$

con

$$F(e, q) = (f_1(e, q), f_2(e, q), \dots, f_n(e, q)),$$

donde $f_i(e, q)$ sería la $f(e, q, q_i)$ de la definición dada).

La función de transición se puede representar mediante una matriz para cada símbolo de entrada:

$$T(e) = \begin{bmatrix} f(e, q_1, q_1) & f(e, q_1, q_2) & \dots & f(e, q_1, q_n) \\ f(e, q_2, q_1) & f(e, q_2, q_2) & \dots & f(e, q_2, q_n) \\ \dots & \dots & \dots & \dots \\ f(e, q_n, q_1) & f(e, q_n, q_2) & \dots & f(e, q_n, q_n) \end{bmatrix}$$

Ejemplo:

$$E = \{a, b, c\}; S = \{0, 1\}; Q = \{q_1, q_2, q_3\}$$

$$h(q_1) = h(q_2) = 0; h(q_3) = 1$$

$$T(a) = \begin{bmatrix} 0, 2 & 0, 7 & 0 \\ 0, 2 & 0 & 0, 1 \\ 0, 6 & 0 & 0, 1 \end{bmatrix}; T(b) = \begin{bmatrix} 0, 7 & 0 & 0, 4 \\ 0, 5 & 0 & 0, 5 \\ 0, 2 & 0 & 0, 6 \end{bmatrix}$$

$$T(c) = \begin{bmatrix} 0 & 0 & 0, 8 \\ 0 & 0, 8 & 0 \\ 0 & 0, 6 & 0, 2 \end{bmatrix}$$

Gráficamente, representaremos al autómata por el diagrama de la figura 5.12.

El concepto recuerda al de autómata probabilista (AP), pero hay dos diferencias importantes:

- Puesto que no se trabaja con probabilidades, no es necesario que las filas de las matrices de transición tengan componentes cuya suma sea la unidad.
- En el AP, para calcular las probabilidades de transición para una cadena $x = e_1 e_2 \dots e_n$ se multiplicaban las sucesivas matrices. Así, si $x = e_1 e_2$

$$f(x, q_i, q_j) = \sum_k [f(e_1, q_i, q_k) \cdot f(e_2, q_k, q_j)]$$

En el borroso, se aplica la regla de composición de funciones:

$$f(x, q_i, q_j) = \max_k [\min(f(e_1, q_i, q_k), f(e_2, q_k, q_j))]$$

lo que equivale a sustituir el producto ordinario de matrices por el producto máximo.

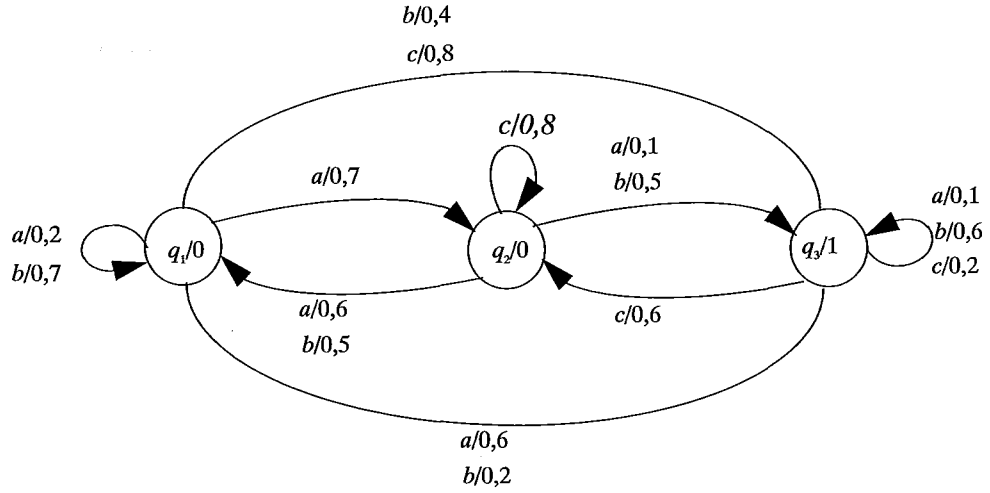


Figura 5.12.

Por ejemplo, la función de transición para la cadena $x = bca$ en el autómata anterior será:

$$\begin{aligned}
 T(bca) &= T(b)T(c)T(a) = \\
 &= \begin{bmatrix} 0,7 & 0 & 0,4 \\ 0,5 & 0 & 0,5 \\ 0,2 & 0 & 0,6 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0,8 \\ 0 & 0,8 & 0 \\ 0 & 0,6 & 0,2 \end{bmatrix} \begin{bmatrix} 0,2 & 0,7 & 0 \\ 0,2 & 0 & 0,1 \\ 0,6 & 0 & 0,1 \end{bmatrix} = \\
 &= \begin{bmatrix} 0,7 & 0 & 0,4 \\ 0,5 & 0 & 0,5 \\ 0,2 & 0 & 0,6 \end{bmatrix} \begin{bmatrix} 0,6 & 0 & 0,1 \\ 0,2 & 0 & 0,1 \\ 0,2 & 0 & 0,1 \end{bmatrix} = \begin{bmatrix} 0,6 & 0 & 0,1 \\ 0,5 & 0 & 0,1 \\ 0,2 & 0 & 0,1 \end{bmatrix}
 \end{aligned}$$

Si inicialmente estamos en el estado q_1 , el subconjunto borroso de Q que expresa el grado de confianza de estar en cada estado será $Q_B^0 = \{q_1/1; q_2/0; q_3/0\}$; tras aplicar la cadena bca , se transformará en

$$Q_B^{bca} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0,6 & 0 & 0,1 \\ 0,5 & 0 & 0,1 \\ 0,2 & 0 & 0,1 \end{bmatrix}$$

$$= \begin{bmatrix} 0,6 & 0 & 0,1 \end{bmatrix} = \{ (q_1/0, 6); (q_3/0, 1) \}$$

Y lo mismo que con el autómata probabilista, puede definirse un *autómata borroso de estructura variable*, $ABEV = \langle E, S, Q, f, h, A \rangle$, donde A es el *esquema de refuerzo o de actualización*, que genera f_{i+1} en función de f_i, e_i, s_i , y utilizarlo en un entorno aleatorio como sistema de aprendizaje. La ventaja con relación al APEV es que los esquemas de actualización están basados en productos máx-mín de matrices, lo que le confiere mayor velocidad. El principal inconveniente es la dificultad para establecer estudios analíticos sobre las propiedades de convergencia.

7. Autómatas celulares

Evidentemente, los AF, sean deterministas, estocásticos o borrosos, pueden interconectarse: si tenemos dos AF (deterministas, para simplificar),

$$A_1 = \langle E_1, S_1, Q_1, f_1, g_1 \rangle$$

y

$$A_2 = \langle E_2, S_2, Q_2, f_2, g_2 \rangle$$

tales que $E_2 = S_1$, y si suponemos que A_2 siempre recibe como entrada el símbolo de salida generado por A_1 , el conjunto es un AF:

$$A_{12} = \langle E_1, S_2, Q, f, g \rangle$$

con $Q = Q_1 \times Q_2$ y funciones de transición y salida, f y g , tales que

$$\begin{aligned} f(e, \langle q_1, q_2 \rangle) &= \langle f_1(e, q_1), f_2(g_1(e, q_1), q_2) \rangle \\ g(e, \langle q_1, q_2 \rangle) &= g_2(g_1(e, q_1), q_2) \end{aligned}$$

donde $\langle q_1, q_2 \rangle \in Q$

Como ejercicio (muy sencillo), el lector puede construir el AF que resulta de conectar la salida del sumador binario (figura 2.8) a la entrada del detector de paridad (figura 2.6).

Por supuesto, si también ocurre que $E_1 = S_2$, se puede asimismo conectar la salida de A_2 a la entrada de A_1 (resultando en el comportamiento global fenómenos

típicos de la realimentación). En general, con N autómatas que compartan los mismos alfabetos de entrada y salida podemos construir *redes de autómatas*. Aunque cada uno de los autómatas componentes tenga un comportamiento individual muy sencillo, si N y el número de conexiones son suficientemente grandes, el comportamiento colectivo del sistema puede ser tan complejo como para imitar los fenómenos que se encuentran en los sistemas físicos o biológicos reales.¹

Los autómatas celulares son redes de autómatas en las que:

- Los componentes, todos idénticos, se llaman *células*, y están distribuidos espacialmente (generalmente se considera un espacio de una o dos dimensiones) formando una "rejilla" o "retículo" regular. Esta distribución puede adoptar distintas formas, pero siempre es regular. Por ejemplo, en un espacio de dos dimensiones las células pueden ser "triangulares" (triángulos equiláteros, cada uno de los cuales puede tener hasta otros tres triángulos "adosados"), "cuadradas", "octogonales"...²
- Cada célula es una máquina de Moore con $S = Q$ ($h(q) = q$) y $E = Q_v$, donde Q_v es el "estado de la vecindad": el estado compuesto por los estados de todas las células que rodean a ésta. Como *vecindad* puede entenderse un conjunto más o menos grande de células. Por ejemplo, en un autómata celular de dos dimensiones con células "cuadradas" puede considerarse que la vecindad de una célula está formada por un máximo de otras cuatro células (las que tienen lados adyacentes con ella), o por un máximo de ocho (si se cuentan también las que limitan por las esquinas), o también pueden extenderse el "radio" y considerar parte de la vecindad a células adyacentes a las anteriores.
- La función de transición es la misma para todas las células:

$$f: Q \times Q_v \rightarrow Q$$

y se suele considerar un modo de *funcionamiento síncrono*, es decir, las transiciones tienen lugar de manera simultánea en todas las células.

Aun con células binarias ($Q = \{0,1\}$) y funciones de transición muy sencillas (por ejemplo, la "regla de la mayoría": si el número de "unos" en $\{Q, Q_v\}$ es superior al número de "ceros" entonces el estado siguiente es "1", y en caso contrario es "0") se obtienen configuraciones espacio-temporales complejas. Pero los

¹Esto se debe a un principio de la ciencia de los sistemas que se suele expresar informalmente con una frase atribuida a Aristóteles: "el todo es más que la suma de las partes". Es decir, de la combinación de elementos simples pero ricamente interconectados surge un sistema con modos de comportamiento complejos e impredecibles a partir del comportamiento sencillo de cada uno de los elementos.

²El lector comprenderá fácilmente por qué se les llama también "autómatas de mosaico" ("tessellated automata").

autómatas celulares son especialmente interesantes cuando, además, se les dota de reglas de *reproducción y desaparición*: en función del estado de la vecindad y del suyo propio, una célula puede "morir" o bien, si su vecindad no está completa, "procrear". Resultan entonces modos de comportamiento evolutivos y complejos que guardan relación con un campo de mucha actualidad: los sistemas caóticos y los fractales.

8. Sistemas con aprendizaje

8.1. ¿Puede aprender una máquina?

El campo de la inteligencia artificial llamado "aprendizaje automático", o "aprendizaje en máquinas" está conociendo un gran auge en los últimos años. Este asunto desborda ya, realmente, los objetivos de este libro, pero dado que el capítulo 6 del tema "Lógica" se ha dedicado a la ingeniería del conocimiento, y que en el apartado 5.1 hemos avanzado una definición (provisional) de aprendizaje, parece conveniente que nos detengamos por un momento a apuntar las ideas básicas del campo. Lo haremos de manera muy esquemática y nada formalizada, remitiendo al lector interesado a la selección bibliográfica del apartado 10.

La opinión de que la capacidad de aprender es una característica indispensable para poder considerar a un sistema como "inteligente" parece bastante razonable. Según lo dicho en el apartado 5.1, sólo se puede afirmar que un sistema aprende si se modifica a sí mismo en el curso de su interacción con el entorno, generando cambios adaptativos en su estructura ("adaptativos" significa que los cambios tienen por finalidad que el sistema responda mejor en ese entorno). Si aceptamos estas dos premisas, entonces es discutible que la mayoría de los sistemas artificiales llamados "inteligentes" (y, concretamente, los sistemas expertos) lo sean realmente: ¿significa la transferencia de conocimientos de un experto humano al sistema un "cambio adaptativo" en el sistema? Más bien parece que no, de la misma manera que no consideramos que un ordenador "aprenda" al cargar un programa en su memoria.

La pregunta que encabeza este apartado³ no es ajena al debate sobre la inteligencia artificial mencionado al principio del capítulo 6 del tema "Lógica". Ahora bien, considerando el comportamiento de algunos sistemas uno se siente inclinado a responder afirmativamente. Así, el conocido como "NetTalk" recibe a la entrada secuencias de caracteres alfabéticos y produce a la salida sonidos. En una fase de *entrenamiento* se le van presentando frases escritas junto con su locución correcta ("ejemplos"), y el sistema va ajustando ciertos parámetros

³ Entiéndase por "máquina" un "sistema artificial", a fin de evitar una respuesta similar a la que daba Claude Shannon ante la pregunta de si las máquinas pueden pensar: "You bet. I'm a machine and you're a machine, and we both think, don't we?" (Horgan, 1992).

internos (son los "cambios adaptativos"). Después de un número suficientemente grande de ejemplos, el sistema ha "generalizado", en el sentido de que sabe "leer en voz alta" frases que no se le han presentado como ejemplos durante el entrenamiento. Otros sistemas son los que aprenden a reconocer los caracteres manuscritos de una determinada persona (tras una fase de entrenamiento para esa persona), que se están utilizando ya industrialmente, por ejemplo, en los "ordenadores de lápiz" ("pen computers").

Sin embargo, si *aprender es adquirir conocimientos*, la respuesta depende de lo que entendamos por "conocimiento". En los casos mencionados puede entenderse que, tras la fase de entrenamiento, el sistema "tiene el conocimiento" necesario para traducir de caracteres escritos a sonidos en un caso, o para distinguir unos caracteres de otros, en el otro. Pero ¿dónde está este conocimiento, y qué forma tiene? Está en los parámetros internos del sistema, que se han ido ajustando, y su forma, numérica, es ininteligible para un ser humano. "NetTalk", o un sistema similar, podrá diferenciar entre, y pronunciar más o menos aceptablemente, "this" y "these", pero de ninguna manera puede decirse que haya *aprendido* el concepto de "fonema"; un sistema de reconocimiento de caracteres podrá llegar a distinguir entre "A" y "H", pero no nos informará de la diferencia morfológica entre ambos.

Ahora bien, como veremos más adelante, también hay procedimientos que permiten *adquirir y formar conceptos*. Es posible, por ejemplo, diseñar un programa que obtenga descripciones estructurales imprecisas de los distintos caracteres escritos ("concepto H" = "dos segmentos aproximadamente verticales unidos en las proximidades de sus puntos medios por un segmento más o menos horizontal"). Sistemas de este tipo ya no sólo aprenden: descubren regularidades en los datos de entrada, generalizan y expresan los conceptos aprendidos en un lenguaje simbólico y comprensible.

Así pues, la respuesta, en cualquier caso, es "sí". Un caso es el del aprendizaje entendido como capacidad de cambiar adaptativamente mediante la modificación de ciertos parámetros internos. En el otro caso se supone que el sistema dispone de algún mecanismo de representación del conocimiento de forma inteligible para la mente humana, y los cambios adaptativos consisten en modificar conocimientos previos o adquirir conocimientos nuevos. Diremos que los sistemas que adoptan el primer punto de vista siguen un *enfoque conductista*, y los que se centran en el segundo, un *enfoque cognoscitivo*.

En cuanto a las aplicaciones, aparte de las ya comentadas (síntesis de voz y reconocimiento de caracteres), hay una de tipo práctico en la ingeniería del conocimiento: la adquisición del conocimiento (tema "Lógica", capítulo 6, apartado 1.4). La adquisición automática o semiautomática para la construcción sistemas expertos es una alternativa al procedimiento "tradicional" de entrevistar a expertos humanos. El sistema puede inducir sus propias reglas generales a partir de "ejemplos" concretos (casos resueltos por el experto o con solución conocida por cualquier otra vía). Otra aplicación de gran interés y potencialidad para la

evolución futura de la ingeniería del conocimiento es la posibilidad de inducción automática de conocimientos explorando grandes bases de datos (apartado 8.4).

8.2. Enfoques conductistas: redes neuronales

Las primeras propuestas de sistemas con aprendizaje automático se basaron en el concepto de *refuerzo* de la psicología conductista. El "sistema aprendiz" ha de ser susceptible de sufrir modificaciones estructurales, ante la actividad de un agente externo (el "crítico", o "maestro") que lo *premia* o lo *castiga* en función de sus respuestas, acertadas o no, ante diferentes estímulos. El algoritmo de refuerzo incluido en el sistema determina las modificaciones que corresponden a los premios y los castigos, y si este algoritmo es adecuado el sistema irá mejorando su comportamiento (dando más respuestas acertadas) a medida que se le vayan proporcionando casos. La definición del apartado 5.1 es típicamente conductista, y el esquema de la figura 5.10 obedece a este enfoque. El "maestro" de la descripción anterior haría la función de evaluación, y también podría proporcionar los estímulos; el esquema de actualización, o algoritmo de refuerzo, sería algo "cableado" o "programado" en el propio sistema que actuaría de acuerdo con el resultado (premio o castigo) de la evaluación.

Los autómatas de aprendizaje responden a este enfoque conductista. Y también las *redes neuronales*, que son, actualmente, los sistemas con enfoque conductista más utilizados y con más aplicaciones desarrolladas. Veamos sus principios.

Las redes neuronales recuerdan algo a los autómatas celulares, pero se diferencian de ellos en dos cosas:

- Las "células", ahora llamadas *neuronas*, son elementos combinacionales (sin memoria).
- No existe una "configuración espacial". Las redes se forman conectando la salida de cada neurona a entradas de otras, de manera totalmente irregular o bien por *capas*; en este segundo caso, hay una capa de neuronas de entrada que recibe los estímulos y cuyas salidas se conectan a las neuronas de una segunda capa, y así sucesivamente, hasta la capa de neuronas de salida.

Un modelo básico de neurona es el elemento lógico con umbral, con n entradas ("sinapsis") y una salida ("axón") binarias, y cada entrada puede estar ponderada por un *peso*, w_p , comprendido entre -1 y +1; la salida es "1" si la suma de las entradas, multiplicada cada una por su peso, supera el umbral, Θ , y "0" en caso contrario.

El *Perceptron* es una red neuronal diseñada para la clasificación de formas gráficas. La entrada, o "retina", está formada por células sensibles a la luz, o *células sensoriales*, cuyas salidas (con valor "0" o "1", dependiendo de la intensidad luminosa en el punto) se conectan directamente (sin pesos intermedios) a entradas de neuronas llamadas *células asociadoras*, y la salida de éstas a cada una de las *células de respuesta* de la salida a través de pesos ajustables. No todas las

células sensoriales se conectan a todas las asociadoras. La selección de conexiones corresponde a lo que en reconocimiento de formas se llama *extracción de características*, y en el *Perceptron* se establece de manera aleatoria. La figura 5.13 ilustra el esquema en el caso de que sólo haya una neurona de respuesta. En este caso sólo es posible una decisión binaria; por ejemplo, decidir si la forma presente a la entrada es de tipo rectangular o no (con n neuronas de respuesta se pueden establecer 2^n clases). Antes de seguir adelante, digamos que este sencillo sistema (con una sola neurona, o, de manera más general, con una sola capa de neuronas: las células sensoriales y las asociadoras, que carecen de pesos, no se considera que formen capas) no es capaz de realizar esa tarea (es decir, de aprender a distinguir una forma rectangular de otra que no lo sea); sí es posible resolverla con redes de más de una capa.

El mecanismo para el aprendizaje en el *Perceptron* es un algoritmo que modifica los pesos durante la *fase de entrenamiento*. Por ejemplo, un algoritmo muy sencillo para el caso de salida binaria de la figura 5.13 consiste en actuar solamente en caso de error (algoritmo de sólo castigo), del siguiente modo: si para una determinada forma de entrada la salida es "0" cuando debería haber sido "1", aumentar los pesos correspondientes a las conexiones con neuronas asociadoras cuya salida es "1", y si la salida es "1" cuando debería haber sido "0", reducir los pesos. El objetivo es que, tras un determinado número de ejemplos en la fase de entrenamiento, el sistema sea después capaz de clasificar sin errores, o con un porcentaje de errores aceptable. Normalmente se parte de un conjunto de ejemplos clasificados correctamente que se particiona en dos, un *conjunto de entrenamiento* y un *conjunto de comprobación*; con los elementos del primero se ajustan los pesos mediante el algoritmo de refuerzo, y el segundo sirve para evaluar en qué medida el sistema ha aprendido. La extensión del algoritmo para n neuronas de respuesta es inmediata: cada una tiene sus propios pesos, y se entrena independientemente de las otras.

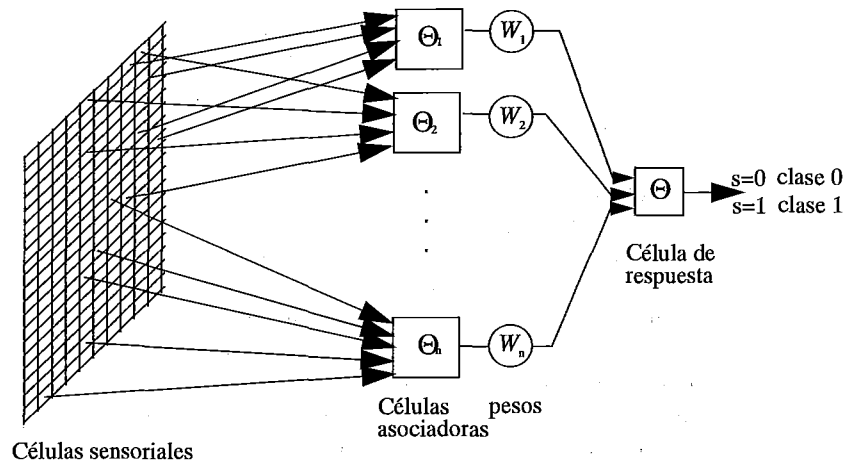


Figura 5.13.

Este sistema, con una sola capa, tiene una limitación grave: la condición para que el sistema converja durante el entrenamiento hasta conseguir un conjunto de pesos que discriminen perfectamente entre las clases es que haya *separabilidad lineal* en el espacio de características (espacio con tantas dimensiones como células asociadoras, en el que se proyectan los ejemplares presentados a la entrada). Esto quiere decir que para cada clase debe existir un hiperplano que separe a las proyecciones de los elementos de esa clase de las de todos los demás. Esta condición es extraordinariamente restrictiva, pero basta con introducir dos "capas ocultas" para formar regiones de separación arbitrarias en el espacio de características. (Una neurona oculta es aquella cuya salida no se toma como salida del sistema, sino que activa a otra u otras neuronas a través de nuevos pesos). El algoritmo anteriormente esbozado ya no es válido (¿cómo podemos evaluar las salidas de las neuronas ocultas?), y se han desarrollado otros más complejos. Mencionemos solamente, para terminar este apunte sobre redes neuronales, el nombre del más conocido: *algoritmo de retropropagación*.

Las redes neuronales son, al menos en su origen, un modelo "biónico" (es decir, un modelo para ingeniería inspirado por los sistemas biológicos) basado en la *ontogenia*: la formación de un sistema nervioso mediante modificaciones de las conexiones sinápticas para adaptarse al entorno. Otro tipo de modelos biónicos son los basados en la filogenia, o evolución de las especies. Son los *algoritmos genéticos*, y los *programas evolutivos*, en los que el sistema está formado por una "población" con una "dotación genética" que va evolucionando por mutación, entrecruzamiento, y otros *operadores genéticos*. También son ya numerosas las aplicaciones en ingeniería de estos sistemas, y de nuevo remitimos al lector a la bibliografía del apartado 10 si está interesado en su estudio.

8.3. Enfoques cognoscitivos: adquisición de conceptos

Durante los años 60 se produjo un cambio de paradigma dentro de la psicología, pasando del "conductismo" al "cognitivismo". Simplificando mucho, y utilizando términos familiares en informática e ingeniería, la psicología conductista estudia la mente como una "caja negra", mientras que la cognoscitiva se centra en las estructuras y procesos internos mentales. Las redes semánticas y los marcos (tema "Lógica", capítulo 6, apartado 4) son ejemplos de modelos procedentes de la psicología cognoscitiva.

En los enfoques cognoscitivos al aprendizaje automático el énfasis no se pone tanto en la mejora del rendimiento del sistema (medido como porcentaje de respuestas correctas) como en que éste sea capaz de construir o de refinar estructuras de conocimiento (sentencias lógicas, redes semánticas, etc.). La figura 5.14 ilustra este nuevo enfoque: las acciones del sistema se basan en estructuras almacenadas en una base de conocimientos, y el aprendizaje del sistema consiste en construir o modificar tales estructuras. Las estructuras de representación del

conocimiento estarán expresadas en un *lenguaje simbólico*. De ahí que algunos autores llamen *simbólicos* a los sistemas basados en estos enfoques, y *subsimbólicos* a los basados en enfoques conductistas. A estos últimos, como generalmente tienen una estructura formada por muchas unidades elementales con muchas conexiones entre ellas, se les llama también *sistemas conexionistas*.

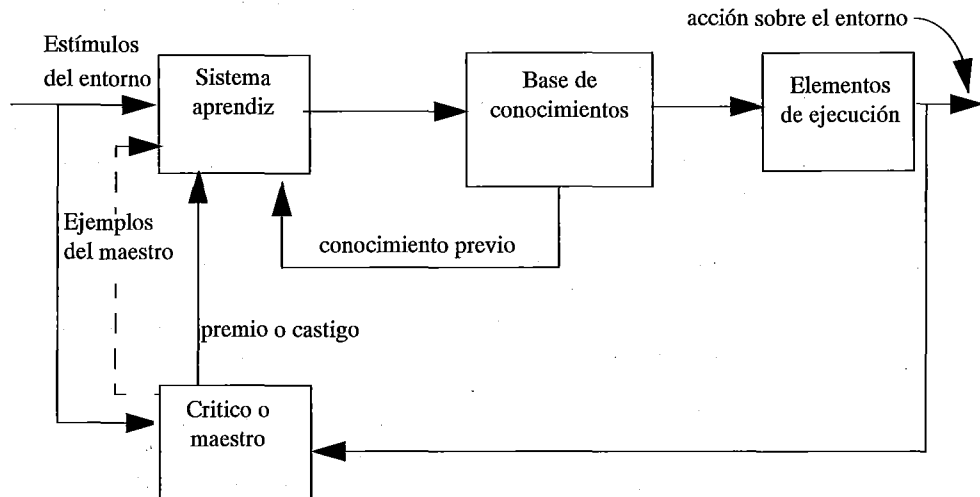


Figura 5.14.

Hay diferentes aproximaciones para la realización de sistemas de aprendizaje cognoscitivos, que podemos clasificar teniendo en cuenta tres dimensiones: el lenguaje de descripción, la estrategia de inferencia y el conocimiento de base.

El *lenguaje de descripción* tiene dos componentes, una para describir a los objetos (o sea, los ejemplos que se le suministran al sistema para que aprenda y los que luego se le presentan para que aplique el conocimiento adquirido) y otra para describir el conocimiento. Los sistemas más conocidos utilizan algún lenguaje de descripción por *atributos valorados*, es decir, los objetos se describen dando los valores que toman sus propiedades o atributos, y representan el conocimiento mediante árboles o reglas de decisión. Estos lenguajes son del tipo "0+" que comentábamos en el apartado 2.5 del capítulo 6 del tema "Lógica".

La *estrategia* más común es, como en los enfoques conductistas, de tipo *inductivo*: a partir de ejemplos, se induce una descripción general. Pero también hay estrategias puramente *deductivas*. Así, un tipo de aprendizaje que recientemente está siendo muy estudiado es el "basado en explicación", que consiste en transformar deductivamente la definición de un concepto en otra definición más operativa por medio de la explicación de un solo ejemplo de ese concepto. Otro tipo, que combina elementos inductivos y deductivos, es el *aprendizaje por analogía*, en el que tiene lugar una transferencia de conocimientos (por ejemplo,

una solución conocida de un problema) a una situación nueva (por ejemplo, un problema similar).

En cuanto al *conocimiento de base* (conocimiento que existe en el sistema previamente a la fase de aprendizaje), en los métodos puramente inductivos puede ser nulo, mientras que en los deductivos y analógicos, por su propia naturaleza, es esencial. Pero también en el aprendizaje inductivo puede ser importante. En efecto, el proceso de aprendizaje es asimilable a una búsqueda en un espacio de descripciones, y el conocimiento de base puede servir para que esa búsqueda no sea "ciega", sino guiada. Esto está muy estrechamente ligado con el lenguaje de descripción. Antes hemos dicho que en muchos sistemas el lenguaje es del tipo "0+", lo que limita el rango de problemas que pueden tratarse: por ejemplo, no pueden aprenderse conceptos que se refieran a objetos estructurados o a relaciones entre objetos; para ello sería preciso utilizar un lenguaje basado en la lógica de predicados. El problema de utilizar un lenguaje de descripción más expresivo es que el espacio de búsqueda se hace también mucho más extenso y aparece el fantasma conocido en la inteligencia artificial como *explosión combinatoria* (fenómeno que ya hemos comentado en el tema "Lógica": apartados 4.8 del capítulo 4 y 2.5 del capítulo 6). De manera más rigurosa, y utilizando un término que se definirá con precisión en el capítulo 7 del tema "Algoritmos", podemos decir que el uso de estos lenguajes conduce a problemas de *complejidad superior a la polinómica* en los algoritmos de aprendizaje. Es aquí donde entra en juego el conocimiento de base, "podando" la búsqueda y reduciendo la complejidad de modo que el problema se haga tratable.

En cualquier caso, los sistemas más conocidos son los llamados de *aprendizaje empírico*. Son sistemas inductivos sin conocimiento de base en los que el problema del aprendizaje se expresa en términos de *adquisición de concepto*. Un concepto es, en principio, una clase de equivalencia. Los ejemplos que forman el conjunto de entrenamiento deben estar *clasificados* en un número finito de clases, y la tarea del sistema consiste en adquirir a partir de ellos una descripción para cada clase, de modo que cuando se presente la descripción de un objeto nuevo pueda asignársele una de las clases. Como ejemplo de sistema de este tipo, uno ya clásico es la aplicación que se hizo de un algoritmo de aprendizaje empírico llamado "AQ" a un sistema experto para el diagnóstico de enfermedades en la planta de la soja. El número de enfermedades distintas (conceptos) era quince, y el de síntomas (atributos) treinta y cinco. El sistema experto había sido construido inicialmente con técnicas tradicionales de adquisición del conocimiento (entrevistas con expertos para la obtención de reglas), y su precisión en el diagnóstico era 71,8%. A lo largo del trabajo se habían llegado a recopilar 630 casos de enfermedades diagnosticadas; de ellos se tomaron al azar 290 (conjunto de entrenamiento) sobre los que se aplicó el algoritmo "AQ", y las reglas obtenidas se comprobaron con los 340 casos restantes (conjunto de comprobación). La precisión obtenida al sustituir las reglas de los expertos por las inducidas fue 97,6%.

8.4. Formación de conceptos y descubrimiento

Hasta ahora hemos venido suponiendo implícitamente, tanto en las figuras como en el texto, que existe un "crítico" o "maestro" que "enseña" al sistema, bien sea porque le da "premios" y/o "castigos" durante la fase de entrenamiento (como suele hacerse en los enfoques conductistas), o porque ha proporcionado previamente un conjunto de ejemplos "clasificados" con los que el sistema trabaja luego para inducir su conocimiento (como en "AQ"). Se dice en todos estos casos que el aprendizaje es "supervisado". En el aprendizaje "no supervisado" *no* hay maestro: el sistema debe descubrir por sí mismo regularidades en los datos de entrada e inducir categorías o conceptos. El aprendizaje supervisado se llama también *adquisición de conceptos*, y el no supervisado, *formación de conceptos*. En formación de conceptos hay tres campos: agrupamiento conceptual, descubrimiento cuantitativo y descubrimiento cualitativo.

Los sistemas de *agrupamiento conceptual* buscan "similitudes" entre los objetos, los agrupan en clases (o conceptos) y obtienen descripciones de estas clases. Su utilidad radica en que pueden descubrir clasificaciones cuando el número de objetos y de sus atributos y relaciones es muy grande para la intuición humana. Con el *descubrimiento cuantitativo* se trata de encontrar relaciones numéricas en los datos. Así, el sistema "BACON" ha "redescubierto" muchas leyes clásicas de la física y la química. El *descubrimiento cualitativo* pretende encontrar entre los datos relaciones más generales, descubrir conceptos y formar teorías.

Aparte de su evidente interés científico, estos sistemas tienen una aplicación práctica inmediata en los sistemas de información. En efecto, es un hecho sobradamente conocido el crecimiento exponencial del volumen de datos almacenados en los servidores de información repartidos por todo el mundo y las posibilidades de acceso cada vez mayores a esos datos. Los sistemas de descubrimiento automático pueden aplicarse a facilitar la navegación por esas fuentes y a encontrar en esas grandes masas de datos la información que se busca (y que se esconde en ellos). A este campo se le llama también "*minería de datos*".

9. Resumen

El modelo básico de autómatas finitos (AF) descrito en los anteriores capítulos puede generalizarse en varias direcciones. Las redes de Petri (RdP) y los modelos derivados de ellas son generalizaciones útiles porque permiten representar, de forma más expresiva que con AF, sistemas discretos compuestos por subsistemas que se comunican entre sí. Esto tiene gran interés, por ejemplo, en el procesamiento distribuido, en las redes de ordenadores y en los protocolos de comunicaciones.

Una vía alternativa para la modelación de sistemas complejos son los autómatas celulares, formados por retículos de células simples en un espacio geométrico, y que encuentran aplicaciones, por ejemplo, en procesamiento de imágenes.

Otra generalización surge al considerar modelos estocásticos o borrosos. En principio, su interés es la modelación de sistemas en los que no se conocen con precisión las relaciones causales. Pero también son la base para el diseño, mediante autómatas de estructura variable, de sistemas que "aprenden" (mejoran su comportamiento) al ir interactuando con su entorno.

Y este último asunto nos ha conducido al campo del aprendizaje automático entendido de un modo más general. Aquí solamente hemos esbozado las ideas básicas del campo. Como compensación, en el apartado siguiente nos extendemos algo más de lo habitual tanto en su evolución histórica como en referencias bibliográficas para el lector interesado en estudiarlo con detalle.

10. Notas histórica y bibliográfica

Las RdP proceden del modelo propuesto por Carl Adam Petri (1962) en su tesis doctoral. Varios investigadores, a mediados de los 70, las "redescubrieron" como modelo básico para sistemas distribuidos (Peterson, 1977, Bochmann, 1979). Más detalles pueden encontrarse en el artículo de Murata (1989).

Durante los años 40 y 50 von Neumann formuló (entre otras cosas más conocidas) una teoría general de autómatas (von Neumann, 1951) que incluía a los autómatas celulares como modelos de sistemas capaces de autorreproducción (von Neumann, 1966). Holland se basó en estos modelos para proponer la primera arquitectura distribuida de ordenador, el "tessellated computer" (Holland, 1959). En los años 70 y 80 se hicieron muy populares las simulaciones de autómatas celulares bidimensionales con células cuadradas, gracias a un programa de John H. Conway que quizás conozca el lector: el "juego de la vida" (Gardner, 1971-72).

El concepto de autómata estocástico apareció en EE.UU. al principio de los años 60 (Rabin, 1963), pensando más bien en el objetivo de mejorar la fiabilidad de los circuitos secuenciales (Winograd y Cowan, 1963) que en su aplicación a sistemas de aprendizaje. Algo antes, von Neumann (1956) ya había introducido la noción de probabilidad en el modelo de neurona formal de McCulloch y Pitts (1943) para demostrar teóricamente que, gracias a la redundancia, es posible sintetizar sistemas fiables a partir de componentes poco fiables. Pero en ninguno de estos modelos se consideraba aún la posibilidad de autoadaptación de la estructura del sistema.

Aunque hay precedentes aislados, algunos muy interesantes e influyentes, como los programas de Samuel (1959, 1967) que aprendían estrategias para el juego de las damas, el interés por los sistemas de aprendizaje (conductistas) aparece casi simultáneamente al principio de los años 60 en EE.UU. y en lo que antes era la "Unión Soviética". Por una parte, Tsetlin (1961) introdujo la idea de

utilizar AF deterministas en un entorno aleatorio como modelos de aprendizaje, y algo más tarde, Varshavskii y Vorontsova (1963) demostraron que el número de estados se reduce utilizando autómatas estocásticos con actualización de las probabilidades de transición (es decir, con estructura variable). Por otro lado, Rosenblatt (1962), propuso el Perceptron, introduciendo mecanismos de refuerzo en una sencilla red de neuronas. La formulación del autómata borroso aparece algo más tarde (Wee y Fu, 1969). En esta definición, que es la que hemos presentado en el apartado 6, la borrosidad sólo se da en la función de transición; Virant y Zimic (1995) dan una formulación más general, con conjuntos de entradas, estados y salidas borrosos.

Por su evidente analogía con los sistemas nerviosos naturales y por su potencialidad para el aprendizaje en general (la "retina" puede generalizarse de modo que las entradas sean señales de cualquier naturaleza) el Perceptron despertó mucho interés hasta que Minsky y Papert (1969) demostraron sus limitaciones, y, haciendo hincapié especialmente en la separabilidad lineal, pronosticaron muy poco futuro para ellos, incluso para las redes multicapa. Su libro fue muy influyente, y los 70 y primeros 80 fueron años "oscuros" para las redes neuronales, pero es interesante indicar que los trabajos de Widrow (1962), paralelos a los de Rosenblatt, derivaron en sistemas adaptativos con aplicaciones importantes en sistemas de control y procesamiento de señales (Widrow y Lehr, 1990).

La dificultad para encontrar algoritmos de aprendizaje en redes multicapa se resolvió, en dos líneas independientes, con el algoritmo de retropropagación de Rumelhart, Hinton y Williams (1986) y con la máquina de Boltzmann de Hinton y Sejnowski (1986), ambos publicados en un libro (Rumelhart y McClelland, 1986) que señala claramente el despegue actual de trabajos sobre el tema. La aplicación "NetTalk" mencionada en el apartado 8.1 se describe en Sejnowski y Rosenberg (1986).

Los primeros algoritmos genéticos los propuso Friedberg (1958, 1959): partiendo de una colección de programas o de estructuras de datos en un momento ("población"), generaba mutaciones aleatoriamente sobre algunos de ellos, seleccionaba los mejores (basándose en algún criterio de "adaptación" dependiente de la aplicación) y los recombinaba entre sí, obteniendo así la siguiente "generación", sobre la que volvía a iniciar el proceso. Los resultados fueron decepcionantes, como los posteriores de Fogel *et al.* (1966) (que acuñó la expresión "programación evolutiva"): hacían falta millones de generaciones para conseguir algo mínimamente alentador.⁴ Holland (1975) retomó la idea, pero con una variación sustancial: la mutación no debe ser el único mecanismo, ni siquiera el fundamental; sólo se aplica esporádicamente, y se introducen otros operadores genéticos, como el trueque de genes ("cross-over"). Tales operadores se aplican a cadenas

⁴Los autores atribuyeron la causa del fracaso al uso del lenguaje de máquina y probaron mutaciones en el nivel de organigrama; hoy, mejor conocida la problemática de la búsqueda, sabemos que la estrategia "ciega" de "genera y comprueba" conduce a explosión combinatoria y al problema de los mínimos locales.

binarias en un sistema en el que las cadenas representan individuos que compiten por "sobrevivir". Este nuevo enfoque ha demostrado su utilidad en diversas aplicaciones de control y optimización.

Los primeros sistemas de algoritmos de aprendizaje siguiendo un enfoque cognoscitivo se basaron en algunos modelos, pobres en formalización pero muy ricos conceptualmente, propuestos dentro de la psicología en los años 60. A partir del "CLS" (Concept Learning System) de Hunt *et al.* (1966), Quinlan formalizó e implementó el primero y más conocido de los algoritmos de aprendizaje cognoscitivo, el "ID3" (Quinlan, 1979, 1986), que ha servido de base para muchos sistemas y aplicaciones. ID3 induce árboles de decisión a partir de ejemplos, y a resultados similares llega AQ con un lenguaje de descripción sobre una lógica "0+" (Michalski, 1975). La descripción de la aplicación basada en AQ para el diagnóstico de enfermedades de la planta de la soja se encuentra en Michalski y Chilausky (1980). Los sistemas de descubrimiento más conocidos son "BACON" de Langley *et al.* (1986), "AM" (Lenat, 1977) y "EURISKO" (Lenat, 1983), de los que han derivado muchas investigaciones posteriores cuyas referencias pueden encontrarse en los libros que citamos más adelante.

Los distintos modelos que hemos mencionado se originaron de manera independiente, pero más recientemente se han empezado a estudiar las relaciones entre unos y otros y las posibilidades de utilizarlos de manera sinérgica. Así, se introducen funciones borrosas en redes neuronales (Kosko, 1991; Simpson, 1992, 1993) y en redes de Petri (Cao y Sanderson, 1995), se combinan algoritmos genéticos y autómatas celulares (Mitchell *et al.*, 1993), se aplican redes neuronales al diseño de sistemas expertos ("sistemas expertos conexionistas": Gallant, 1988), se usan redes neuronales borrosas para la generación de reglas (Mitra y Pal, 1995), etc. Las referencias anteriores tienen un interés esencialmente histórico, y las hemos incluido para dar una panorámica sobre la evolución de los modelos tratados en este capítulo. Para estudiarlos con detalle recomendamos los siguientes libros:

Silva (1985) y Reisig (1985) tratan las redes de Petri y sus aplicaciones, Gupta *et al.* (1977) los autómatas borrosos, y Narendra y Thathachar (1989) los autómatas estocásticos y de aprendizaje. Aplicaciones de éstos (son especialmente interesantes las que se refieren al encaminamiento adaptativo en redes de comunicaciones) se describen en el capítulo 9 de este último libro, y en varios capítulos del de Glorioso (1980). Los autómatas celulares y sus aplicaciones pueden estudiarse en Wolfram (1986), Toffoli y Margolus (1987) y Gutowitz (1991).

En redes neuronales, la explosión de publicaciones desde 1986 ha sido tal que resulta ya difícil seguir los avances. Es muy recomendable el libro de Anderson y Rosenfeld (1988), que recopila las publicaciones pioneras más importantes. Un tratado sistemático es el de Hertz *et al.* (1991) y un libro sobre sus aplicaciones a sistemas expertos el de Gallant (1993). Una buena perspectiva puede conseguirse con los artículos publicados en dos números monográficos del "Proceedings of the I.E.E.E." (septiembre y octubre de 1990). El I.E.E.E. publica también un "Trans-

actions on Neural Networks" desde 1990, y desde 1988 Pergamon Press publica "Neural Networks".

Sobre algoritmos genéticos hay menos literatura, pero también crece rápidamente. Están los libros de Goldberg (1989) y de Michalewicz (1992), los trabajos de síntesis de Holland (1986) y DeJong (1990) y las actas de los congresos que se celebran regularmente desde 1985: Grefenstette (1985, 1987), Schaffer (1989), Belew y Booker (1991), Rawlings (1991) y Whitley (1992). El libro de Holland *et al.* (1986) trata de inducción en general, pero presenta los algoritmos genéticos tal como los introdujo modernamente Holland. Para una perspectiva sobre el tema son recomendables los artículos de Srinivas y Patnaik (1994) y Ribeiro *et al.* (1994).

Sobre psicología cognoscitiva (o "cognitiva", como prefieren los especialistas) hay, naturalmente, multitud de textos. Nosotros hemos encontrado especialmente útil y esclarecedor el de Vega (1985).

El libro de Soucek (1991) contiene una colección de trabajos sobre integración de paradigmas (redes neuronales, algoritmos genéticos, sistemas borrosos, etc.) para la construcción de sistemas inteligentes, y el de Gallant (1993) trata de sistemas expertos conexionistas.

La "biblia" del aprendizaje en máquinas la forman cuatro volúmenes titulados "Machine learning: an artificial intelligence approach" (Michalski *et al.*, 1983, 1986; Kodratoff y Michalski, 1990; Michalski y Tecucci, 1994). En ellos pueden encontrarse trabajos de síntesis escritos por los autores más conocidos de los distintos enfoques del aprendizaje. El libro de Kodratoff (1988) cubre todos los enfoques cognoscitivos utilizando fundamentalmente un lenguaje lógico, y el compilado por Carbonell (1989) tiene un alcance más reducido, pero incluye capítulos sobre redes neuronales y algoritmos genéticos. Un estudio teórico del aprendizaje puede encontrarse en el libro de Osherson *et al.* (1986).⁵ Desde 1986 se publica la revista "Machine Learning" (Kluwer Academic Publishers, Holanda). Sobre sistemas de descubrimiento en general trata el libro de Ziarko (1994), y en cuanto a su aplicación en bases de datos, hay que acudir a las recopilaciones de las comunicaciones presentadas a los congresos sobre el tema: Piatetsky-Shapiro y Frawley (1991), Piatetsky-Shapiro (1993) y Fayyad y Uthurusamy (1994).

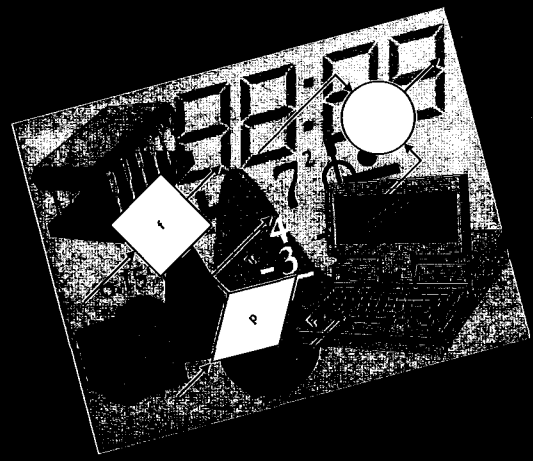
⁵ Para hacerse una idea de su enfoque basta con esta frase del prefacio: "Tal como se desarrolla en este libro, la teoría del aprendizaje una colección de teoremas sobre ciertos tipos de funciones de la teoría de números".

Parte III

Algoritmos

Fundamentos de informática

1



Ideas generales

1. Algoritmos y ordenadores

La palabra *algoritmo* procede del apellido latinizado de un matemático árabe, Mohamed ibn Mûsâ al-Khowârizmî (o al-Khârezmi) que en 828 y 825 d.C. escribió respectivamente dos tratados, el primero de cálculo con los números hindúes y el segundo de resolución de ecuaciones. La deformación del título de esta última obra ha originado el nombre de álgebra, con el que se conoce a la rama de las matemáticas consagrada al cálculo literal.

En nuestro siglo, no solamente se ha hecho más frecuente el uso del término *algoritmo*, sino que su contenido, su significación precisa, ha sido explorado profundamente. En la técnica de los computadores es palabra de uso cotidiano, aunque es necesario advertir que no todos los profesionales de la informática la conocen y emplean en su sentido cabal.

Lo cierto es que la aparición de los ordenadores con sus extraordinarias posibilidades de proceso y almacenamiento de información ha impulsado fuertemente el estudio de los algoritmos. Tanto es así, que a veces es difícil percibir que

pueden distinguirse *dos campos de interés*: uno, de índole fundamental en matemáticas, en donde se plantea el problema de *si tienen solución ciertos tipos de problemas, lo que es equivalente a preguntarse si existe un algoritmo que conduzca siempre a una solución para esa clase de problema*. En el próximo capítulo se abordarán las definiciones formales del concepto de algoritmo, pero por el momento podemos aceptar que un algoritmo es, intuitivamente, la expresión de una secuencia precisa de operaciones que conduce a la resolución de un problema.

El otro campo de interés investiga o afronta *los problemas que surgen de la aplicación de las máquinas computadoras al procesamiento de algoritmos*. En este campo caen, entre otras, las relaciones de los algoritmos con las estructuras de los datos y con los lenguajes de programación, y la complejidad computacional (o algorítmica) es decir, la cantidad de recursos necesaria para computar determinados problemas. El estudio de la complejidad algorítmica ha adquirido en las dos últimas décadas un nivel teórico considerable.

Puesto que hablamos de resolver problemas, será bueno tener una idea general de los principales aspectos implicados en dicha tarea. La resolución de un problema implica un proceso de varias etapas, que, a grandes rasgos, son las expresadas en el flujograma de la figura 1.1.

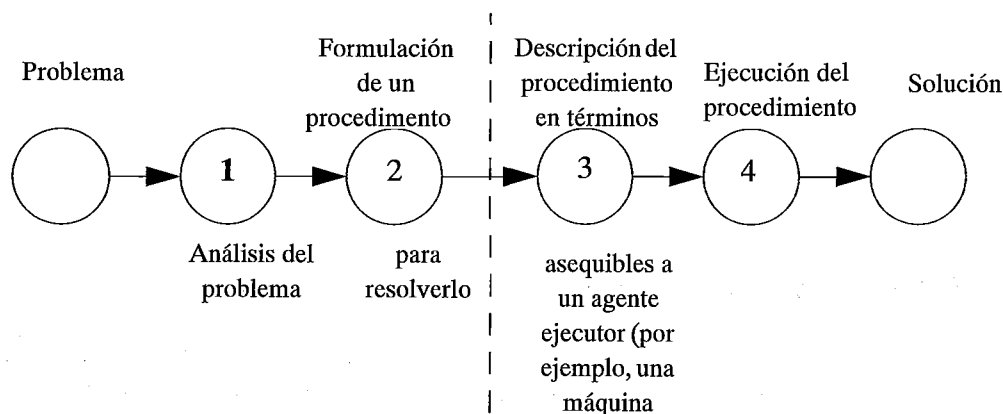


Figura 1.1.

Si el problema abordado tiene solución, la descripción del procedimiento para llegar a ésta estará fuertemente teñida por las características interpretativas y operativas del agente ejecutor del procedimiento que, en nuestro caso, será normalmente una máquina, y en particular un ordenador.

Así pues, el momento del proceso que se señala con una línea vertical de trazos marca la frontera habitual desde donde la principal preocupación del autor del algoritmo empieza a ser su comunicación con la máquina. Cuanto más evolucionada sea ésta, menor será el esfuerzo del autor. De hecho, podemos esperar que la etapa 3 pueda desglosarse en una secuencia 3', 3'', 3''', 3'''' tal que las reglas

descriptivas en un lenguaje sean transformadas por una máquina M' en reglas descriptivas en otro lenguaje, etc., sucesivamente, hasta llegar a una máquina que sea capaz de ejecutar dicha reglas.

Por la misma razón, las etapas 2 y 3 pueden verse fundidas si se es capaz de formular el procedimiento directamente en términos de reglas o instrucciones para una máquina. En definitiva, la secuencia $3', 3'', 3'''$, ..., supone contar con la ayuda de un encadenamiento de algoritmos de transformación de expresiones simbólicas, y, por consiguiente, la repetición otras tantas veces, y a niveles y para problemas distintos, del proceso 1-2-3-4. Resumiendo, *el grado de concentración o multiplicación de las fases dependerá del conocimiento del sujeto acerca del problema a resolver y de la operatividad de los recursos ejecutores de que disponga.*

2. Algoritmos, lenguajes y programas

La referencia anterior al papel del lenguaje en la expresión de la secuencia de operaciones que constituye el algoritmo hace inevitable entrar a considerar las relaciones existentes entre los tres términos que dan título a este apartado.

Con un programa, escrito en un lenguaje concreto, expresamos nuestro algoritmo para ser procesado por una máquina concreta, provista del adecuado procesador de lenguaje. De manera que *un programa para ordenador es la expresión de un algoritmo en un lenguaje artificial formalizado.*

Hablábamos en el primer apartado del esfuerzo necesario para describirle un algoritmo a un ordenador. La noción de lenguaje pone de manifiesto en forma nítida cómo el grado de ese esfuerzo, entre otros factores importantes, es función de la potencia operativa del lenguaje y de su grado de sintonía funcional con las clases de operadores implicadas en el algoritmo en cuestión. El siguiente ejemplo permitirá visualizar de una manera clara esta cuestión. Se trata de un programa para calcular el máximo común denominador de dos números A y B , siguiendo el conocido algoritmo de Euclides. Las expresiones del programa difieren a veces considerablemente, en su forma y volumen, según el lenguaje utilizado. Si empleamos el lenguaje PL/I obtenemos el siguiente programa:

```
IF A = 0 THEN
  ULTIMO: DO;
    MCD = B; RETURN; END;
IF B = 0 THEN DO;
  MCD = A; RETURN; END;
AQUI: G = A/B;
/*Suponiendo G variable entera*/
R = A - B * G;
IF R = 0 THEN GO TO ULTIMO;
A = B; B = R; GO TO AQUI;
```

Este mismo algoritmo, descrito en un lenguaje de máquina o en un lenguaje ensamblador, sería mucho más largo y en cierta manera incomprensible para una mente humana no entrenada. Si le aplicásemos un lenguaje inapropiado como DYNAMO, que es un lenguaje de simulación de sistemas continuos, la expresión de este algoritmo podría hacerse prácticamente imposible o por lo menos extraordinariamente enrevesada. En cambio, resultaría muy sencilla de escribir y de comprender si contásemos con una máquina capaz de entender y ejecutar la instrucción CALL MAX (). Este programa consistiría simplemente en la sentencia CALL MAX (A, B, MCD), que presupone la existencia de una máquina especializada, con dos entradas A y B para los dos números a procesar y una salida MCD para el resultado.

Es evidente que no siempre se dispone de una máquina así y cuando se dispone de ella no podríamos asegurar que tenga existencia física en la forma que sugiere la figura 1.2. Se dispondría más bien de una máquina virtual que, en términos generales, es el ordenador armado con un programa procesador de lenguaje de unas determinadas potencia y funcionalidad. En tales circunstancias, *el autor del algoritmo puede desentenderse de las características de la máquina física y expresarlo en un formato adecuado a las características funcionales de la máquina virtual*, esto es, *del lenguaje escogido*. Esta idea puede tal vez expresarse de otra manera diciendo que el lenguaje transforma el ordenador en una máquina virtual.

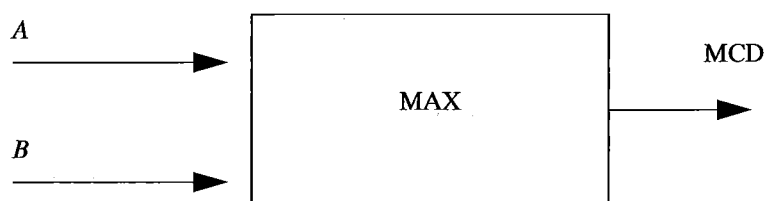


Figura 1.2.

Aunque la programación estructurada es una técnica ya muy conocida, se ha decidido incluirla en este tema porque nos proporciona al menos tres elementos didácticos importantes. En primer lugar, y en relación con lo que se acaba de decir, este estilo de programación, integrado en un diseño descendente, desarrolla la noción de *recurso abstracto*, conceptualmente similar en la práctica a la de máquina virtual. En segundo lugar, el ejercicio profesional de la programación adolece de serios problemas y deficiencias, tales como costes elevados, falta de fiabilidad, dificultades de mantenimiento y otros, sobre los que un buen estilo de programar ejerce una influencia positiva. Y, por último, el origen de la programación estructurada, debido a un trabajo de Böhm y Jacopini publicado en 1966, está conectado al concepto de autómatas y, en particular, al de máquina de Turing, artefacto central en el tratamiento de este tema en este libro. A la programación estructurada le dedicaremos un capítulo.

3. Algoritmos y máquinas de Turing

En el año 1936, antes del advenimiento de los ordenadores, el matemático inglés A. M. Turing inventó una máquina computadora de una increíble simplicidad en su estructura lógica. El concepto de algoritmo puede muy bien estudiarse en términos de esquemas funcionales de máquinas de Turing, ya que, como veremos, éstas permiten disecar los algoritmos en una secuencia de las operaciones más elementales que cabe imaginar.

La máquina de Turing se ha convertido en *la piedra angular de la moderna teoría de algoritmos*. Siendo una máquina ideal, que cada uno define y construye con papel y lápiz, no se ve afectada por los progresos tecnológicos. Es, pues, un invariante de la informática y también un símbolo. La asociación de informática profesional más prestigiosa, la A.C.M. (Association for Computing Machinery), entrega todos los años el premio Turing a uno de los científicos entre quienes más hayan contribuido a generar avances significativos en el dominio de la informática.

Con la máquina de Turing dispondremos de un artefacto teórico con el que formalizar el concepto de algoritmo a través del concepto de *función computable*: un problema computable es un problema para el que existe un algoritmo. En su forma más sencilla, la máquina de Turing es un autómata finito que controla una cinta infinita. Véase en la figura 1.3 cómo, por afán de hacerlo más tangible y próximo a las máquinas físicas, se ilustra gráficamente este autómata en un artículo de Wang, de 1965, en la revista "Scientific American".

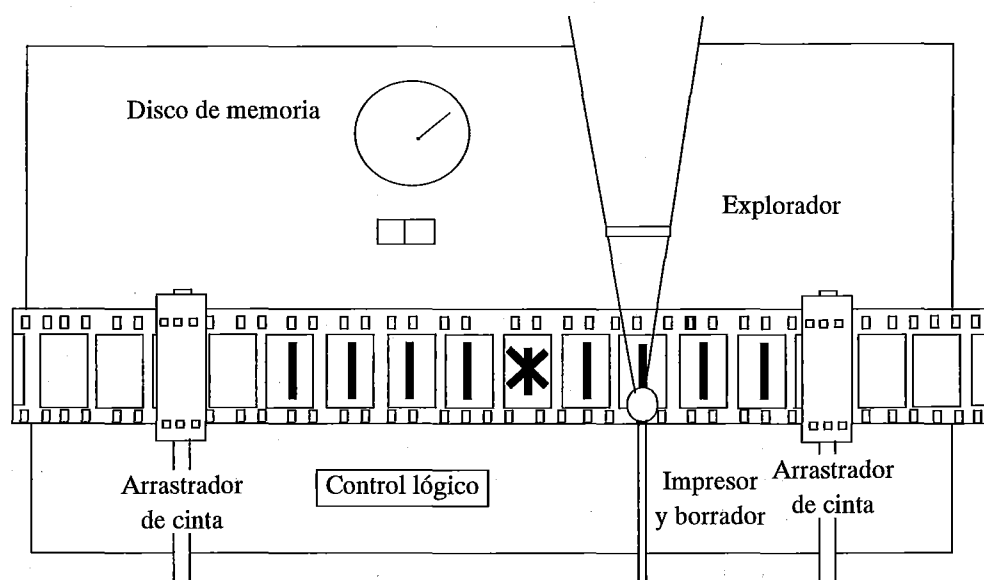


Figura 1.3.

Además de desarrollar algunas cuestiones de importancia teórica relacionadas con las máquinas de Turing y los algoritmos, habremos de enfatizar el análisis y el diseño de estas máquinas como una vía hacia la comprensión de la complejidad de la tarea de diseñar algoritmos cuando las máquinas virtuales son de muy bajo nivel (en otras palabras, cuando los lenguajes son de muy bajo poder expresivo) y de cómo, con un lenguaje tan elemental como el de la máquina de Turing, se puede expresar cualquier algoritmo. En tal sentido, existe una relación de la máquina de Turing con la complejidad algorítmica. Se dedicarán los capítulos 5 y 6 y algún apartado del séptimo a todas estas cuestiones.

No acaba aquí el interés de la máquina de Turing. Tratándose de un autómata, podría habérsela considerado en el tema "Autómatas". Sin embargo, su particular relevancia le ha hecho merecedora de estos capítulos específicos. Por parecidas razones, la relación de los autómatas con los lenguajes, brevemente abordada ya en "Autómatas", será ampliada en el tema de "Lenguajes". Ciertos autómatas son capaces de reconocer las cadenas de símbolos que constituyen el lenguaje generado por una determinada gramática. Las máquinas de Turing reconocen lenguajes generados por sistemas de escritura no restringidos, a los que se llama lenguajes de tipo 0. En el capítulo 6 encontraremos una breve consideración de este punto, a la espera de sistematizarlo en "Lenguajes".

4. Computabilidad y complejidad

Hemos dicho que un problema computable (o decidible) es aquel que admite solución algorítmica. *En los problemas computables es interesante estimar el orden de magnitud de los recursos computacionales* que requieren los distintos algoritmos que puedan resolverlos. La complejidad computacional es, en pocas palabras, ese orden de magnitud. Así, un problema concreto podría necesitar 500 años de cómputo continuado, por lo que, siendo un problema computable, lo consideraríamos un problema no factible, un problema de una complejidad intratable.

Los recursos habituales para procesar algoritmos son: tiempo, capacidad de memoria y velocidad de procesamiento. Por simplificar, fijémonos en el tiempo que, además, depende en alguna proporción de los otros dos factores señalados. Es evidente que, una vez fijado un algoritmo para un determinado problema, el tiempo de cálculo necesario depende del tamaño de los datos del problema. Por ejemplo, multiplicar dos números lleva más tiempo si éstos constan de 1.543 dígitos cada uno que si sólo constan de 10, y lo mismo puede decirse si de lo que se trata es de clasificar un conjunto de elementos: el tiempo es función del tamaño de los datos. La búsqueda de expresiones matemáticas para la estimación del tiempo de procesamiento de un algoritmo como función del tamaño de los datos de entrada entra en el campo de competencia de la teoría de la complejidad, que, entre otras actividades, evalúa por consiguiente el grado de factibilidad de los

problemas. De esta manera se llega a clasificar los algoritmos por la expresión de su complejidad (comportamiento asintótico) en algoritmos de complejidad polinómica y exponencial. Sobre tales cuestiones hablaremos en el capítulo 7.

El lector debe comprender que el estudio de la complejidad algorítmica, pese a haber alcanzado un notable nivel teórico, no puede de ninguna manera ser catalogado como un campo especulativo, puesto que se ocupa de cuestiones eminentemente prácticas o proporciona herramientas para abordarlas. Imagínese un caso como el siguiente: para controlar por computador un reactor nuclear resulta vital conocer el máximo lapso con el que el algoritmo programado es capaz de responder a ciertos supuestos de emergencia. O el caso de tener que elegir un algoritmo cuya complejidad esté acotada entre un límite superior y uno inferior. O el de tener que evaluar la ventaja obtenida entre procesar un algoritmo en un ordenador de estructura secuencial o en otro de estructura paralela.

Otro tipo de complejidad, menos fundamental en sus implicaciones que el anterior, aunque de considerable importancia práctica, es *el de complejidad del software*. Le dedicaremos algún espacio en el mismo capítulo 7. En primera aproximación, esta complejidad tiene que ver con la estructura de predicados del programa construido para describir un determinado algoritmo. Cuanto más rica sea esta estructura parece intuitivamente evidente que la tarea del programador resulta más difícil, mayor su esfuerzo para desarrollarla y también mayor la probabilidad de error, puesto que se acrecienta el número de caminos de proceso que pueden tener que recorrer los datos tratados por el algoritmo. Existe un vínculo de proporcionalidad entre el número de predicados y el número de discriminaciones mentales del programador en su diseño y descripción del algoritmo. La fiabilidad y el coste del software guardan una fuerte relación con la riqueza de tal estructura.

Una forma de visualizar y hasta de medir esta complejidad consiste en asociar a la estructura de control de los programas un grafo orientado. Tomemos el ejemplo sencillo del programa en PL/I para calcular el m.c.d. de dos números, visto en el apartado 1.

La figura 1.4 es el grafo correspondiente a este programa, si suponemos que cada uno de sus nodos representa un bloque de código con flujo de control secuencial y los arcos representan ramas en el programa. No vamos a entrar en detalles ahora. Con lo dicho, cualquiera puede comprender que la dificultad asociada con el diseño y codificación de un programa ha de tener una relación con la magnitud y enrevesamiento del grafo. Esta es a grandes rasgos la complejidad del software. Al mismo tiempo se hace patente que dicha complejidad es función del nivel de lenguaje, puesto que la instrucción CALL MAX (), que resuelve el mismo problema, tiene un grafo absolutamente elemental.

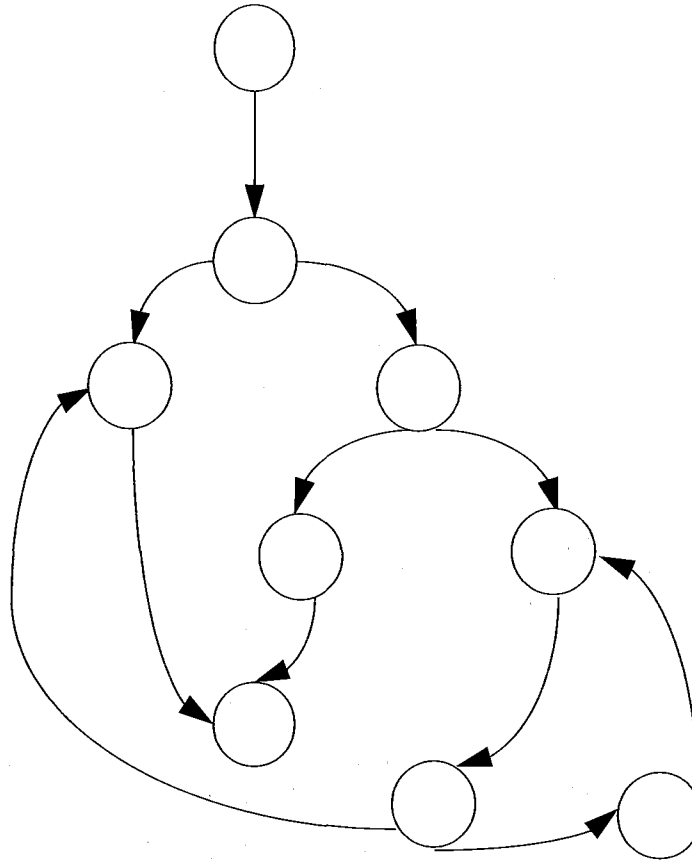


Figura 1.4.

5. Resumen y conclusiones

No es fácil de resumir este capítulo. El esquema conceptual de la figura 1.5 intenta plasmar una imagen incompleta e imperfecta de la íntima relación entre algunos de los conceptos de que versa este libro, pero se pretende que sirva sobre todo como un plano-guía para los siguientes capítulos de este tema.

El sentido de la flecha quiere significar algo parecido al sentido de una navegación conceptual. Así pues, partiendo del concepto de algoritmo, nos vemos conducidos al concepto de máquina ejecutora, bien sea un autómata, bien sea un ordenador, y éste último es básicamente un conjunto de autómatas. Un autómata muy especial, que resulta ser el ordenador universal, es la máquina de Turing, por cuyo intermedio se construye una teoría formal de los algoritmos y se llega a los conceptos de recursividad y computabilidad.

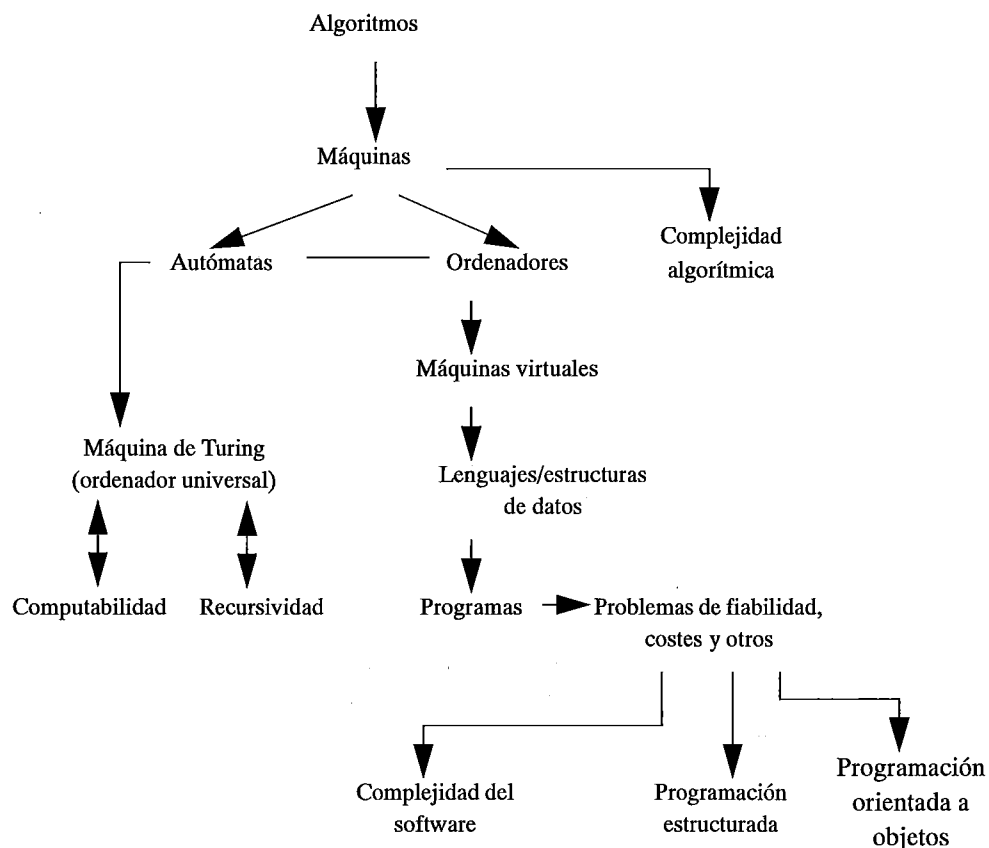


Figura 1.5.

Los ordenadores llevan a la máquina virtual y al lenguaje (incluyendo las estructuras de datos) como niveles convenientes para describir procesos ejecutivos de los algoritmos. En cierta manera, y tal como ha escrito Wirth, "el propio ordenador consiste tan sólo en algoritmos y estructuras de datos" y éste es, en esencia, el núcleo de la naturaleza protéica (multiforme, multiuso) del ordenador. Lenguajes formales y autómatas se corresponden, siendo los distintos tipos de éstos, máquinas reconocedoras de aquéllos, y por tanto, piezas básicas de los programas denominados procesadores de lenguaje (traductores, intérpretes y otros).

Los programas son en definitiva algoritmos descritos en un lenguaje para una máquina, que puede ser un ordenador o una máquina de Turing, por citar dos ejemplos extremos. Por cualquiera de los dos caminos nos encontramos con el problema de la complejidad algorítmica, entendida como cantidad de recursos computacionales necesarios para culminar la ejecución del algoritmo.

En el primer caso, además, tropezamos con problemas de costes y de fiabilidad. La programación estructurada es un método o estilo de programación, que, inicialmente justificado por estudios de teoría de autómatas, es representativo de

un conjunto de técnicas ya clásicas para mejorar los resultados de la programación secuencial. La programación estructurada es el modelo básico de lo que ahora se conoce como *programación imperativa tradicional*. Existe otro modelo de programación llamado programación declarativa, en el que se distinguen "dos estilos" principales, la programación funcional y la programación lógica (véase capítulo 5 del tema "Lenguajes". Pero ciñéndonos ahora a la programación imperativa, ésta ha visto desarrollarse un enfoque muy potente y de gran futuro, la programación orientada a objetos (o programación mediante objetos).

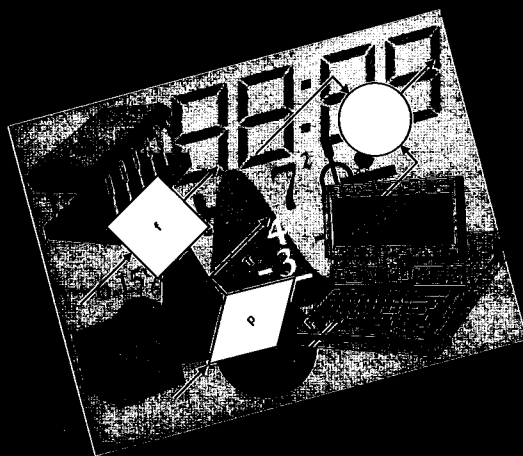
Los mismos problemas han conducido a interesarse por definir y medir la complejidad de los programas. Naturalmente, la complejidad del software tiene relación con los algoritmos a través de la programación, del lenguaje y del nivel de máquina virtual disponible en el ordenador (siguiendo el esquema en sentido inverso), pero no tiene en principio una relación directa con el concepto de complejidad algorítmica. Para muchos autores, esta última complejidad constituye una parcela importante de la teoría de la programación, aunque otros le dan un carácter todavía más básico. En cambio, la complejidad del software es indiscutiblemente un aspecto práctico, metodológico, de la programación.

Un detalle final aparentemente anecdótico, pero que puede dar una cierta medida de la importancia informática atribuida a los conceptos aquí esbozados es que la contribución a su estudio ha proporcionado el premio Turing a varios investigadores, a saber: Knuth (algoritmos, estructura de datos), Dijkstra (teoría de la programación), Wirth (lenguajes), Hoare (algoritmos, teoría de la programación) y Cook (complejidad algorítmica).

El premio Turing del año 1985 se entregó a Karp por sus trabajos sobre complejidad en problemas de tipo combinatorio.

Fundamentos de informática

2



2

Algoritmos

1. Introducción

El presente capítulo se dedica básicamente a definir el concepto de algoritmo. Primero empieza con unas definiciones diversas y no totalmente coincidentes, que se contrastarán. Se pasará a continuación a una definición formal en teoría de conjuntos, que comprende y precisa todas las anteriores.

De la definición se extraen unas propiedades o condiciones que debe cumplir todo algoritmo, a las que se añaden propiedades que se puede desear que cumplan los "buenos algoritmos".

El capítulo prepara ya el terreno para hablar de programas y de máquinas, destacando en la definición formal aquellos elementos, como el número de orden de una ejecución y la existencia misma de los estados, que prefiguran la existencia de determinadas condiciones generales en cualquier máquina ejecutora de algoritmos.

2. Definiciones de algoritmo

No hay una, sino muchas definiciones de algoritmo. Aquí recuadramos varias de distintos autores, todas (¡y no es casualidad!) tomadas de libros sobre computación o computadores.

Definiciones de algoritmo

1. Lista de instrucciones que especifican una secuencia de operaciones que darán la contestación a cualquier problema de un tipo determinado.
2. Conjunto de reglas que define de manera precisa una secuencia de operaciones tales que cada regla es efectiva y definida y tal que la secuencia termina en un tiempo finito.
3. Sucesión finita de prescripciones potencialmente ejecutables expresadas en un lenguaje definido que estipula cómo ejecutar un cierto encadenamiento de operaciones para resolver todos los problemas de un cierto tipo dado.
4. Sistema de reglas que permiten obtener una salida específica a partir de una entrada específica. Cada paso debe estar definido exactamente, de forma que pueda traducirse a lenguaje de computador.

Lo primero que sorprende es que las definiciones difieren no poco entre sí, aunque tal vez sólo sean las apariencias. Esto deja ya suponer que no son muy precisas. Pero fijémonos en las semejanzas y no en las diferencias.

1. Hay unas *reglas*, o instrucciones, o prescripciones.
2. Tales reglas, instrucciones, etc., especifican una *secuencia* (encadenamiento) *de operaciones* o pasos.
3. Aunque de forma muy implícita (excepto en la cuarta definición), las operaciones se supone han de ser llevadas a cabo por un *agente ejecutor*, máquina o ser vivo, por sí o a través de otros agentes. Agente que es el destinatario de las instrucciones.
4. Más implícitamente aún, pero contenido en las definiciones, se establece que la secuencia de operaciones tiene una *duración*, que podrá ser tan larga como se quiera, pero ha de ser *finita*.

Con estos elementos, que constituyen un común denominador de lo que hemos podido descubrir hasta ahora sobre la conceptualización moderna de los algoritmos, vemos que es posible elaborar algoritmos para resolver muchas clases de problemas. Así, por ejemplo, encontrar el máximo común divisor de dos números enteros, investigar si una palabra determinada figura en una tabla de palabras almacenada en la memoria de un computador, jugar una partida de ajedrez o tornear una pieza complicada.

La amplitud del campo de los problemas es tan grandiosa que cualquiera percibe la dificultad de aprehender la noción de algoritmo si uno se sitúa en medio de la diversidad de los problemas y la diversidad de los agentes ejecutores. Es obligatorio reducir el ámbito de atención e investigar el asunto como un proceso intelectual independiente del problema específico, por una parte. De ahí se desprende que, si bien hay algoritmos numéricos y no numéricos, en última instancia todos pueden reducirse a la especificación de operaciones sobre símbolos. Y por otra, es obligatorio independizarse en lo posible del agente ejecutor de estas operaciones simbólicas y, para ello, una solución ha consistido en definir un agente ejecutor único (veremos que será la máquina de Turing), al cual podrían reducirse en última instancia todos los demás.

3. Algoritmos y máquinas

Fijémonos ahora en la ejecución del procedimiento o algoritmo, con independencia del problema y de la máquina de que se trate, para lo cual daremos unas definiciones más formales que en el apartado anterior.

3.1. El concepto de algoritmo, visto desde la teoría de conjuntos

Se define un algoritmo como una cuádrupla A ,

$$A = (Q, E, S, F) \quad (1)$$

con

- Q: Conjunto de todos los elementos simples y de todas las K-uplas que pueden describir el cálculo.
- E: Subconjunto de Q. Sus elementos son los datos de entrada al proceso de cálculo.
- S: Subconjunto de Q. Sus elementos son los diferentes resultados al término del cálculo.
- F: $Q \rightarrow Q$, aplicación que describe la regla de cálculo propiamente dicha y que, a partir de cualquier elemento q_0 , genera la construcción de una sucesión q_0, q_1, q_2, \dots tal que:

$$q_{i+1} = F(q_i), i \in N \text{ con } q_0 \in E \subset Q \quad (2)$$

Para que A represente un algoritmo, cada sucesión (2) debe ser finita. Así pues, es preciso, como condición necesaria no suficiente, que la función F deje invariante el subconjunto S ; es decir: $\forall s \in S, F(s) = s$. Si la sucesión es finita, su

punto de parada viene dado por el menor índice y para el que $q_i \in S$. No será finita si $\nexists i \in N; F(q_i) \in S$.

Ejemplo: Cálculo del m.c.d. de dos números enteros no negativos n_1 y n_2 . El organigrama de la figura 2.1 expresa un procedimiento conocido para resolver este problema. (P.E. significa "parte entera" de dividir n por n').

Este algoritmo que se expresa en la figura 2.1 en una forma, en parte icónica (diagrama), en parte formal (expresiones matemáticas), puede representarse a través del lenguaje de los conjuntos así:

$$Q : (<n>, <n, n'>, <n, n', r, 1>, <n, p, r, 2>, <p, n', r, 3>), n, n', p, r \in N$$

$$E : (<n, n'>)$$

$$S : (<n>)$$

$$F : F(n_1, n_2) = (n_1, n_2, 0, 1) \text{ si } n_1 > n_2, \text{ si no } F(n_1, n_2) = (n_2, n_1, 0, 1);$$

$$F(n) = (n);$$

$$F(n, n', r, 1) = (n) \text{ si } n' = 0, \text{ si no}$$

$$F(n, n', r, 1) = (n, n', n - n' \times P.E.(n/n'), 2);$$

$$F(n, p, r, 2) = (p, p, r, 3);$$

$$F(p, n', r, 3) = (p, r, r, 1);$$

El signo ; separa las distintas variantes de la regla de cálculo que el lector puede usar para hallar la sucesión q_0, \dots, q_i, \dots partiendo de cualquier pareja de números enteros no negativos, familiarizándose así con la definición que se acaba de dar. Por cierto que dicha definición habría que perfeccionarla en el sentido de que *la aplicación F no implique operaciones que no sepa realizar*. En definitiva, las restricciones que habría que imponer a Q, E, S y F son de naturaleza tal que la cuádrupla A no contenga más que operaciones elementales simples (repare el lector en la relatividad del argumento de simplicidad, siendo la operación más simple la que pueda ejecutar un autómata).

Un ejemplo como el del cálculo del m.c.d. pone de manifiesto, mediante el empleo de los números 1,2,3, etc., la idea de que *una aplicación o un cálculo es una secuencia de aplicaciones o cálculos más elementales y que tal secuencia puede expresarse mediante un número asignado a las órdenes que deben ejecutarse*. De forma general, podría decirse que el estado del algoritmo es un par (a, j) , donde j indica el número de la orden que debe ejecutarse y a es la información que caracteriza el estado del algoritmo cuando hay que ejecutar la orden j (información que comprende combinaciones de datos, resultados intermedios y resultados finales). Así, a cada evaluación de la aplicación $F(q_i) = F(a_i, k) = (a_{i+1}, l)$ se le puede llamar ejecución de la orden k del algoritmo. El conjunto S incluiría la información correspondiente a los resultados finales y el conjunto E , la información de los datos de entrada.

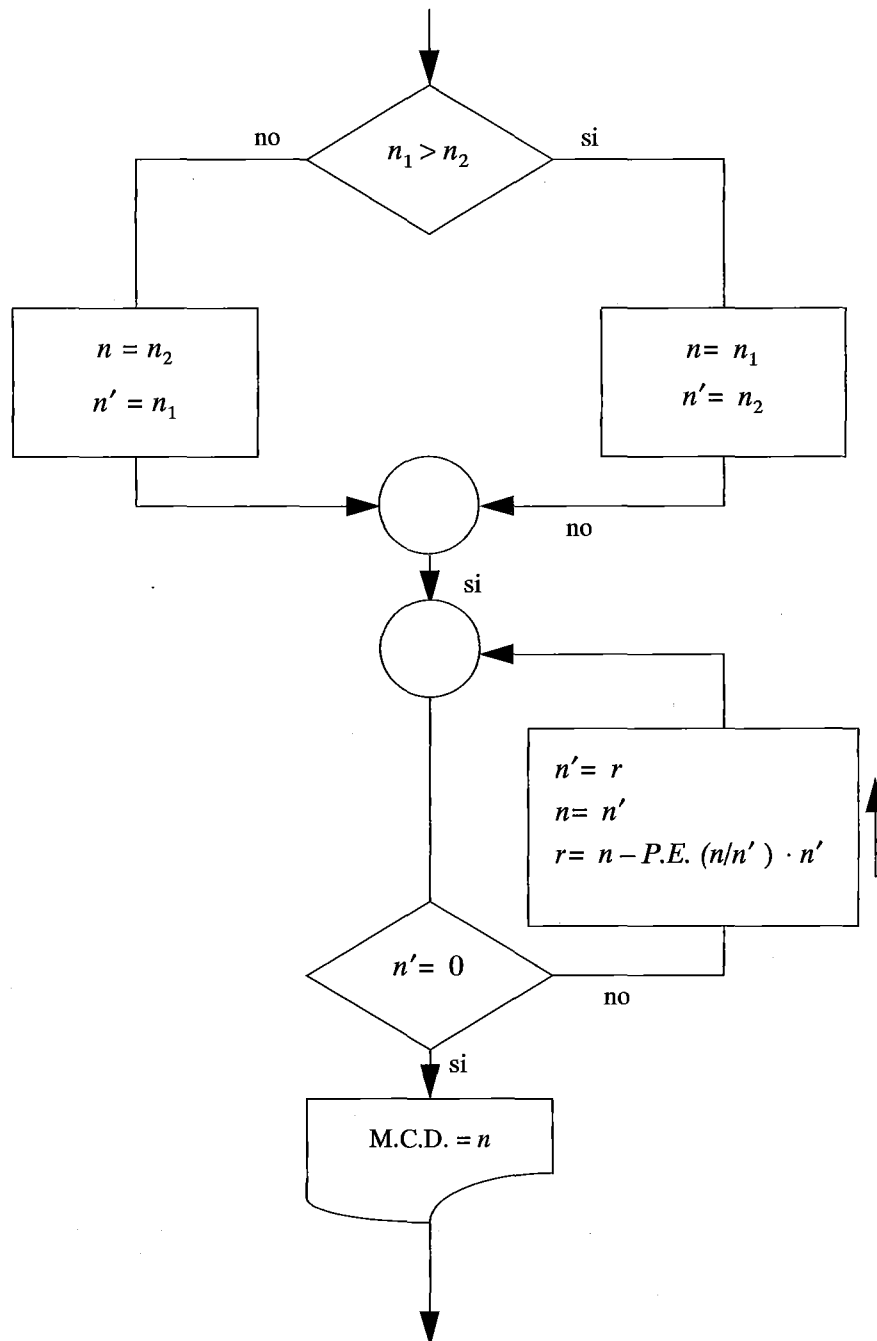


Figura 2.1.

El concepto de algoritmo puede examinarse formalmente también en relación con un alfabeto y en términos de una máquina de Turing, cosa esta última que haremos más adelante. De momento, veamos más de cerca y en forma intuitiva las implicaciones de los algoritmos sobre las máquinas.

3.2. Las máquinas, como estructuras capaces de ejecutar algoritmos

Toda máquina capaz de ejecutar algoritmos (fase 4 del proceso de la figura 1.1) debe tener una estructura con los subsistemas que se destacan en la figura 2.2.

Siguiendo literalmente este enfoque, se definen a continuación los subsistemas básicos de una máquina general ejecutora de algoritmos y el subsistema de memoria que resulta de agrupar todos (o parte) de los elementos de memorización necesarios a la máquina.

3.2.1. Subsistema de entrada/salida (E/S)

Donde se ejecutarán las órdenes de comunicación de la máquina con su mundo exterior.

3.2.2. Subsistema de proceso

Donde se ejecutarán las órdenes de tratamiento de datos que dan lugar a la realización de operaciones del algoritmo.

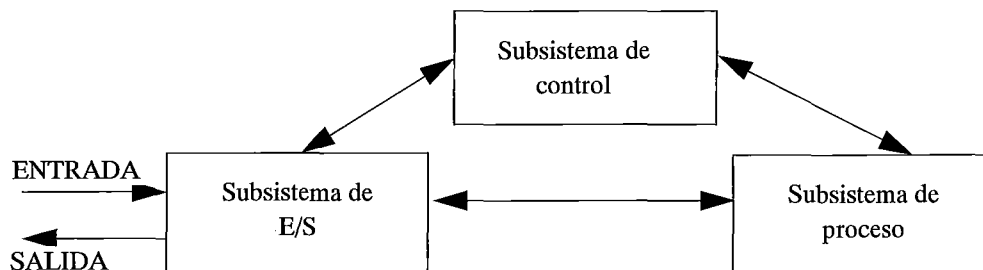


Figura 2.2.

3.2.3. Subsistema de control

Donde se consigue el secuenciamiento adecuado en la ejecución de las órdenes y donde se generan las señales de control adecuadas para el funcionamiento de los subsistemas definidos en los dos apartados anteriores.

Por lo visto en el apartado 3.1, la ejecución de las órdenes de un algoritmo genera un nuevo estado q_{i+1} a partir del estado anterior q_i . Ello implica que hay que memorizar en la máquina el estado del algoritmo. Esto es lo mismo que decir

que los subsistemas podrán llevar asociados elementos de memoria (asociados siempre, como se sabe, a todo circuito secuencial) para memorizar las informaciones de estado que corresponden a la misión de cada subsistema.

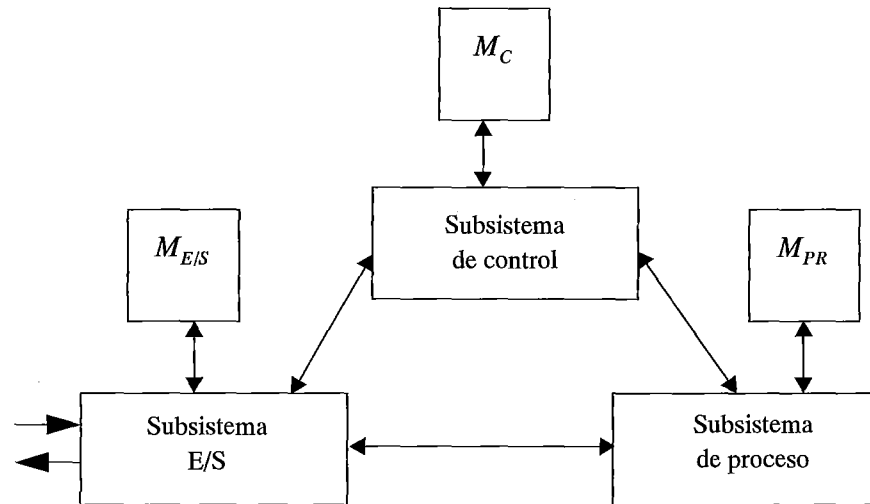


Figura 2.3.

En muchas máquinas (y esto es precisamente lo que ocurre en los ordenadores) resulta más conveniente agrupar todos o parte importante de los mencionados elementos de memorización en un subsistema específico, el subsistema de memoria (figura 2.4).

4. Propiedades de los algoritmos

Como resumen y ampliación de lo dicho, describimos en este apartado las cuatro propiedades que debe poseer todo buen algoritmo.

4.1. Propiedad de finitud

Un algoritmo debe siempre terminarse. Todo algoritmo puede subdividirse en un número de subalgoritmos tan grande como se quiera, pero finito, admitiendo cada uno de estos subalgoritmos cadenas últimas que se terminan.

4.2. Propiedad de definitud

Toda regla de un algoritmo debe definir perfectamente la acción a desarrollar, para aplicarla sin que pueda haber lugar a ambigüedad alguna de interpretación.

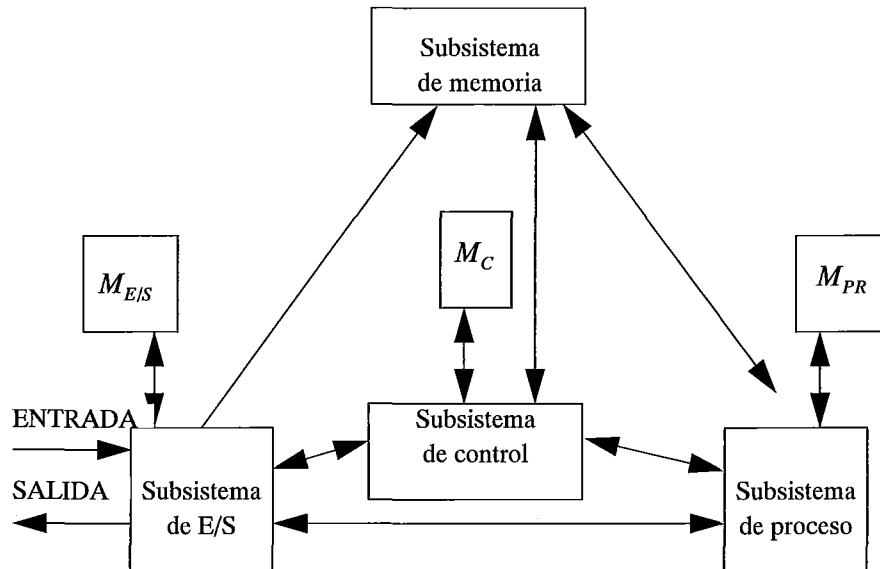


Figura 2.4.

4.3. Propiedad de generalidad

Un algoritmo no debe contentarse con resolver un problema particular aislado sino, por el contrario, toda una *clase de problemas* para los que los datos de entrada y los resultados finales pertenecen respectivamente a conjuntos específicos.

4.4. Propiedad de eficacia

Aun cuando un algoritmo posea las tres propiedades anteriores, se busca mejorarlo por razones de economía, de realizabilidad o de rapidez.

5. Problemas sin algoritmo

Los matemáticos han mostrado históricamente su deseo de resolver tipos de problemas cada vez más generales. Este deseo, como se ha visto, inspira la propiedad de generalidad, que es una propiedad relacionada con el grado de potencia de un algoritmo.

Esto significa, así en abstracto, que sería preferible construir un algoritmo para hallar todas las raíces de una ecuación del tipo

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$$

donde n es un entero positivo arbitrario, a construir un algoritmo para resolver la ecuación $x^n - a = 0$, o, más sencillo todavía, un algoritmo para hallar raíces cuadradas. Ante este planteamiento surge inmediatamente una restricción de orden pragmático en escoger el nivel de generalidad congruente con los propósitos de quien tenga que resolver un problema. También, como se verá, al elevar el nivel de generalidad puede penetrarse en un entorno donde no existan soluciones o, al menos, soluciones conocidas.

Ahora bien, cuando el propósito es de índole teórica, se puede llegar a elevar dicho propósito hasta el nivel del sueño de Leibniz, que consistía en buscar un algoritmo para resolver cualquier problema matemático. Refinado este enunciado, dio en uno de los más famosos problemas de la lógica matemática, *el problema de la deducción*.

Es sabido que, utilizando símbolos, cualquier proposición de una teoría matemática puede escribirse mediante una fórmula y esta fórmula es una palabra definida en un alfabeto. Entonces, la derivación lógica de una proposición se convierte en una cadena de transformaciones de palabras (cálculo lógico). El problema de la deducción se puede formular así:

Para dos palabras cualesquiera (fórmulas) R y S de un cálculo lógico, determinar si existe o no una cadena deductiva de R a S . (S es la proposición y R es la premisa).

Se supone que la solución es un algoritmo para resolver cualquier problema de este tipo. Dicho algoritmo daría un método general para resolver problemas en todas las teorías matemáticas que se construyen de forma axiomática. La validez de cualquier proposición S en tal teoría sólo significa que puede deducirse del sistema de axiomas. Después, la aplicación del algoritmo determinaría si la proposición S era válida o no. Además, si la proposición S fuese válida, entonces podríamos encontrar un encadenamiento deductivo correspondiente en el cálculo lógico y de ahí recuperar un encadenamiento de inferencias que probaría la proposición.

Hasta ahora no se ha encontrado tal algoritmo. De hecho, no se han encontrado algoritmos para problemas menos generales. El matemático alemán Hilbert presentó en 1901, en un congreso que se celebraba en París, una lista de 20 problemas no resueltos. El décimo problema se enunciaba de la forma siguiente: *Hallar un algoritmo para determinar si cualquier ecuación diofántica dada tiene una solución entera.*

Precisamente se conoce un algoritmo válido para resolver una ecuación diofántica con una sola incógnita: $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$, pero no cuando contiene varias incógnitas, como es el caso de la siguiente ecuación: $a^2 + b^2 - c^2 = 0$.

Otro problema que no tiene solución conocida es el de averiguar si dos sucesiones de ceros y unos están relacionadas.

Podemos formar una sucesión de ceros y unos que tenga "descen- dencia" mediante las siguientes reglas: 1. Si la sucesión tiene menos de tres símbolos, parar 2. Si la sucesión comienza por 0, borrar los tres primeros símbolos y añadir 00 al final. 3. Si la sucesión comienza por 1, borrar los tres primeros símbolos y añadir 1101 al final. ¿Hay un algoritmo para determinar si una de las dos sucesiones dadas es descendiente de la otra?	011010001001 01000100100 0010010000 001000000 00000000 0000000 000000 00000 0000 000 00 (parar)	101110110011 1101100111101 11001111011101 011110111011101 11011101110100 111011101001101 0111010011011101 101001101110100 0011011101001101 10110100110100 1101001101001101 (descendencia con- tinua)
---	--	--

Si prescindimos de los detalles operativos relacionados con la ejecución de los algoritmos (detalles de enorme importancia en la práctica) puede decirse: *a) que un problema tiene solución cuando se es capaz de demostrar formalmente que existe un método que genera siempre, a partir de todo elemento del conjunto E, un elemento del conjunto S (expresión (1)) en un número finito de pasos; b) no existe solución cuando se es capaz de demostrar formalmente la inexistencia de algún método que pueda generar un elemento de tal conjunto S en las mismas condiciones expresadas en a). Y, por último, c) no existe solución conocida cuando no se es capaz de demostrar una cosa o la otra.*

6. Resumen

Para empezar *se han definido verbalmente los algoritmos* como un conjunto o lista de reglas, instrucciones o prescripciones que especifican a un agente ejecutor una secuencia finita de operaciones para la resolución de un problema. El problema ha de ser lo más general que sea posible, aunque esta condición es siempre bastante relativa.

Cuando se pretende *resolver un problema, se desarrolla un proceso de varias etapas* en las que normalmente unas son más próximas a la estructura propia del problema (etapas de análisis y de formulación de un procedimiento de solución) y otras más decantadas del lado de las capacidades operativas del (o de los) agente ejecutor (en muchas ocasiones, una máquina y modernamente casi siempre un computador, al menos en nuestro caso). Este proceso resulta tanto más largo o complejo cuanto mayor sea la distancia entre la complejidad del problema y la capacidad operativa del agente ejecutor disponible.

Si se hace caso omiso de este último factor, puede plantearse *una definición formal de algoritmo* como una cuádrupla $A = \langle Q, E, S, F \rangle$, donde la regla de cálculo $F: Q \rightarrow Q$ lleva al algoritmo desde un estado inicial ($\in E$) a un estado

final ($\in S$) en un número finito de pasos, supuesto que se sepa realizar F con todos los elementos de Q .

Pensando ahora precisamente en términos del factor anteriormente descartado, vemos que el uso y el concepto de estado del algoritmo, en donde puede distinguirse un número de orden de ejecución, lleva a concebir un *agente ejecutor-máquina como una estructura con tres o cuatro subsistemas* (entrada/salida, proceso, control y memoria).

No se tiene un algoritmo si el procedimiento elaborado no cumple las condiciones de *finitud* y *definitud*, a las que pueden añadirse, si se quiere hablar de "buenos algoritmos", las de generalidad y eficacia, relacionadas con el grado de ambición o de optimización con que se enfoque la resolución de un problema.

De todas formas *hay problemas que no tienen solución* o, al menos, *solución conocida*. Esta última parte del capítulo nos muestra cómo el grado de generalidad de un algoritmo tiene unos límites y nos pone delante de temas fundamentales de las matemáticas modernas.

7. Notas histórica y bibliográfica

Aparte la mención al origen de la palabra *algoritmo*, creada en homenaje a un matemático árabe, no es fácil establecer una evolución histórica del concepto moderno que tal palabra contiene. Referencias a matemáticos célebres como Leibniz, Hilbert y otros, ya más entrado el presente siglo, nos parecen pertinentes a la redacción de este libro.

Es indudable, y esto conviene resaltarlo de nuevo, que la aparición de los ordenadores hacia la década de los cuarenta impulsó un gran interés por trabajos teóricos anteriores de Turing y otros científicos.

En lo que se refiere a este capítulo, se han empleado los trabajos que a continuación se relacionan.

Las definiciones de algoritmo del apartado 2 se deben por el mismo orden a las siguientes referencias: (Trakhtenbrot, 1960); (Stone, 1972); (Corge, 1975) y (Knuth, 1977).

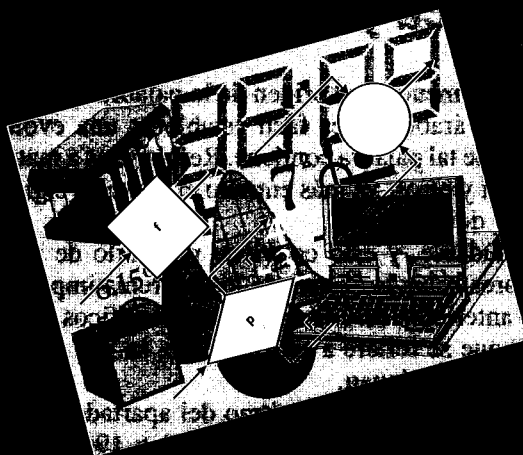
Para la presentación de la relación entre algoritmos y máquinas hemos seguido el planteamiento de Alabau y Figueras (1975).

Las propiedades de los buenos algoritmos se encuentran en Corge (1975).

En cuanto a la formulación del problema de la deducción se ha tomado de Trakhtenbrot (1960) y, por último, el problema de las dos sucesiones de ceros y unos, de H. Wang (1974).

Fundamentos de informática

3



3

Programación estructurada

1. Introducción

La programación estructurada surgió en la década de los sesenta como reacción contra los problemas que para entonces comenzaban a plagar el desarrollo de software, relacionados primordialmente con la *falta de fiabilidad de los programas*, cuya complejidad iba creciendo considerablemente. El *software* de sistemas, por ejemplo (sistema operativo, compiladores, etc), es especialmente complicado, y cualquier fallo conduce a consecuencias, cuando menos desagradables, en muchos casos catastróficas.

Tras imponerse en la década de los setenta, la programación estructurada puede considerarse hoy como un estilo enteramente clásico. Los lenguajes de programación imperativos más utilizados en los últimos años (Pascal, C, Ada) están diseñados de manera que se favorece esta forma de programar, convirtiéndola en algo natural, mientras se desaconseja con énfasis el uso de instrucciones no estructuradas, especialmente *GOTO*.

En un libro como éste no puede faltar un capítulo sobre la programación estructurada, que abarca, en forma muy sumaria, desde aspectos formales básicos sobre programas en general y programas estructurados, hasta cuestiones prácticas sobre diseño y ejemplos de estructuras complementarias facilitadas por lenguajes hoy tan comunes como Pascal y C.

2. Definición formal de un programa para ordenador

En las páginas sucesivas utilizaremos la técnica de los diagramas de flujo (llamados también ordinogramas u organigramas) para describir los algoritmos y los programas de ordenador que los ejecutan (fase 3, figura 1.1, capítulo 1). Aunque esta práctica está en retroceso, ha sido un método muy popular, utilizado ya por los primeros programadores (condesa Ada Lovelace, Adela Goldstine, Grace Hopper, John von Neumann, etc.). Sin embargo, hasta hace poco, la utilización de estos grafos carecía de un soporte riguroso.

2.1. Definición algorítmica

Una primera forma de definir un programa hace uso del concepto de algoritmo, detallado en el capítulo anterior. Dado un algoritmo $A = \langle Q, E, S, F \rangle$ llamamos programa a la cuádrupla

$$P = \langle X, x_0, Y, f \rangle$$

donde

- X : Conjunto de los estados del ordenador.
- $x_0 \in X$: Estado inicial de la máquina, programa cargado y datos iniciales.
- Y : Conjunto de los estados finales de la máquina después del cálculo.
- f : Función de transición de un estado interno al estado siguiente.

Decimos que P representa al algoritmo A si existen tres aplicaciones

- $g: E \rightarrow X$
- $h: X \rightarrow S$, tal que $h(Y) = S$
- $r: X \rightarrow N$

tales que

- El algoritmo A produce el resultado $s \in S$ a partir de $e \in E$ si y sólo si P produce el resultado $y \in Y$ a partir de $g(e)$, de tal modo que $h(y) = s$.

- Si $x \in X$, entonces $F(h(x)) = h(f^* r(x))$, donde la notación $f^* r(x)$ significa que la función f debe ser iterada $r(x)$ veces.

2.2. Función, programa, función de programa

Una *función* es un conjunto de pares ordenados (a,b) tales que

$$(a, b') \in f \wedge (a, b'') \in f \Rightarrow b' = b''$$

Cuando el par $(a, b) \in f$ escribiremos $b = f(a)$, donde a es un argumento de f , b su valor o resultado. El conjunto de todos los argumentos es el dominio de f , y el de todos los valores es la imagen de f .

Llamaremos *estado* del ordenador (y lo representaremos con la letra E) a un conjunto de ternas (m,e,s) , donde m representa el contenido de la memoria del ordenador en un momento determinado, e es el contenido de las colas de entrada y s el de las colas de salida en ese mismo momento. Distinguiremos un estado especial, q_0 , que llamaremos "estado inicial" de la ejecución del programa.

Una *instrucción* es una función que pasa de un estado a otro estado, es decir, modifica el contenido de la memoria, de las colas de entrada o de salida o de varias de estas componentes a la vez.

$$f: E \rightarrow E$$

Naturalmente, una instrucción podría dejar invariable el estado de la máquina. Un *programa* es un conjunto finito de instrucciones

$$P = \{f_1, f_2, f_3, \dots, f_n\}$$

La ejecución del programa es el resultado de aplicar cada una de las funciones o instrucciones que lo componen sobre el estado resultado de la anterior, excepto la primera, que se aplica sobre el estado inicial.

$$q_n = f_n (\dots f_3 (f_2 (f_1 (q_0))) \dots)$$

El valor q_n obtenido se llama estado final de la ejecución.

Se llama función del programa $[P]$ a:

$$[P] = \{(q_0, q_n) \mid q_n \text{ es el estado final si } q_0 \text{ es el inicial}\}$$

El programa P es una regla específica, pero no única, para calcular la función $[P]$. Diremos que dos programas son equivalentes si definen la misma función del programa. Por ejemplo, los programas de la figura 3.1 son equivalentes.

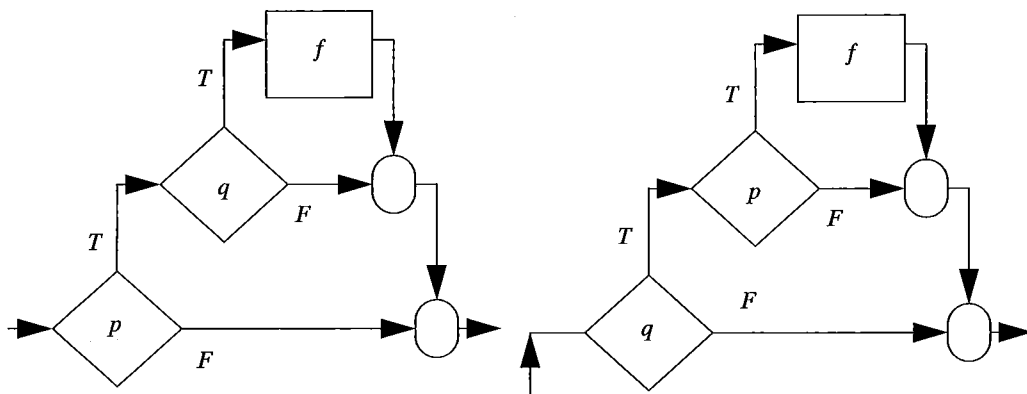


Figura 3.1.

Esta formalización del concepto de "programa" encuentra aplicación práctica para la definición formal de la semántica de lenguajes de programación (Tema "Lenguajes", cap. 5, apartado 4.2).

2.3. Normalización de organigramas

Vamos a formalizar de una manera mejor y más completa la idea de grafo orientado asociado a un programa, que ya se presentó en el apartado 4 del primer capítulo.

Un programa P puede representarse por un grafo $G = (P, U)$ en el que el conjunto de los nodos es el conjunto de instrucciones P y el de los arcos está definido por la siguiente regla:

"Existe un arco desde la instrucción f a la instrucción g si la función g puede aplicarse al estado resultado de f durante la ejecución de P ".

O, lo que es lo mismo, hay un arco de f a g si la instrucción g puede ejecutarse inmediatamente después de la instrucción f .

Este grafo puede adoptar, en último extremo, la forma de un *grafo orientado con tres tipos de nodos* (figura 3.2)

El nodo de función conduce siempre a la misma función siguiente, por lo que tendrá la forma $f(q)$

El nodo de predicado tiene dos arcos de salida, y su definición es:

$$f(q) = \begin{cases} f_1(q) & \text{si } p(q) \text{ es cierto.} \\ f_2(q) & \text{si } p(q) \text{ es falso.} \end{cases}$$

donde $p : E \rightarrow \{\text{True}, \text{False}\}$ ($\{\text{cierto}, \text{falso}\}$) es un predicado que se aplica a un estado y toma el valor True o el valor False. Supondremos que el arco superior o el de la derecha (si el nodo se dibuja verticalmente) corresponde a True, y el inferior o izquierda corresponde a False. (Nota: a veces, en la práctica, se invierte este convenio).

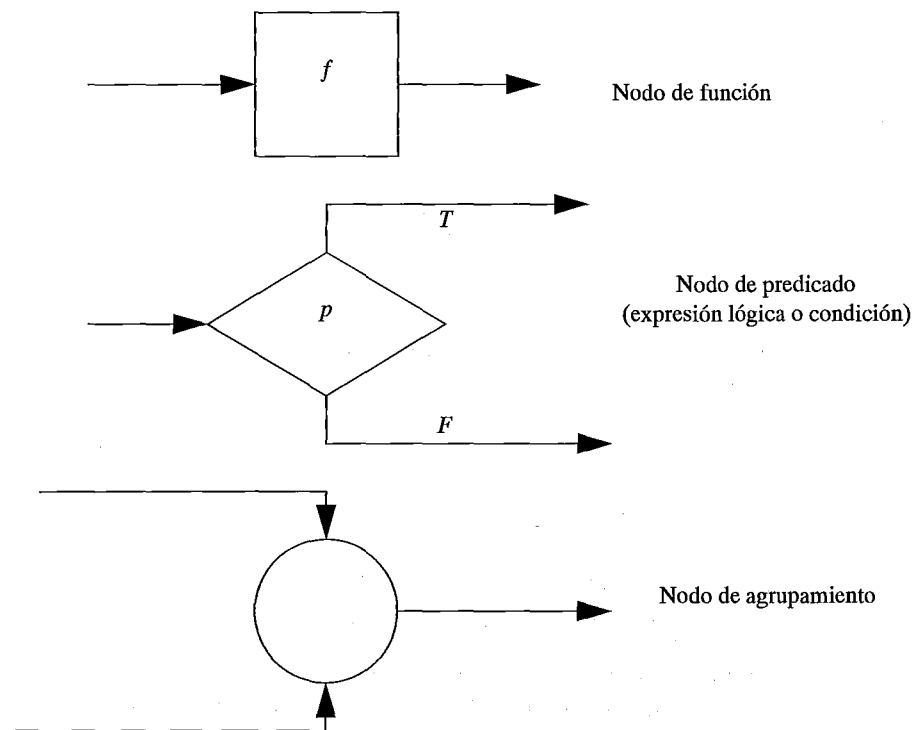


Figura 3.2.

Un nodo de agrupamiento simplemente transfiere control de sus dos líneas de entrada a la de salida.

Definiremos *programa limpio* como aquel cuyo grafo posee solamente un arco de entrada y solamente un arco de salida, existiendo además un camino que lleva desde la entrada hasta cualquier nodo del grafo y desde cualquier nodo hasta la salida.

Se define una clase especial de diagramas BJ (por Böhm y Jacopini) sobre un conjunto de funciones $F = \{f_1, \dots, f_m\}$ y un conjunto de predicados $PR = \{P_1, \dots, P_n\}$ en la forma del cuadro de la figura 3.3. Esta definición es recurrente y permite reconstruir cualquier diagrama de flujo de programas limpios como un diagrama BJ, lo que nos proporciona una forma de normalizar los diagramas de flujo.

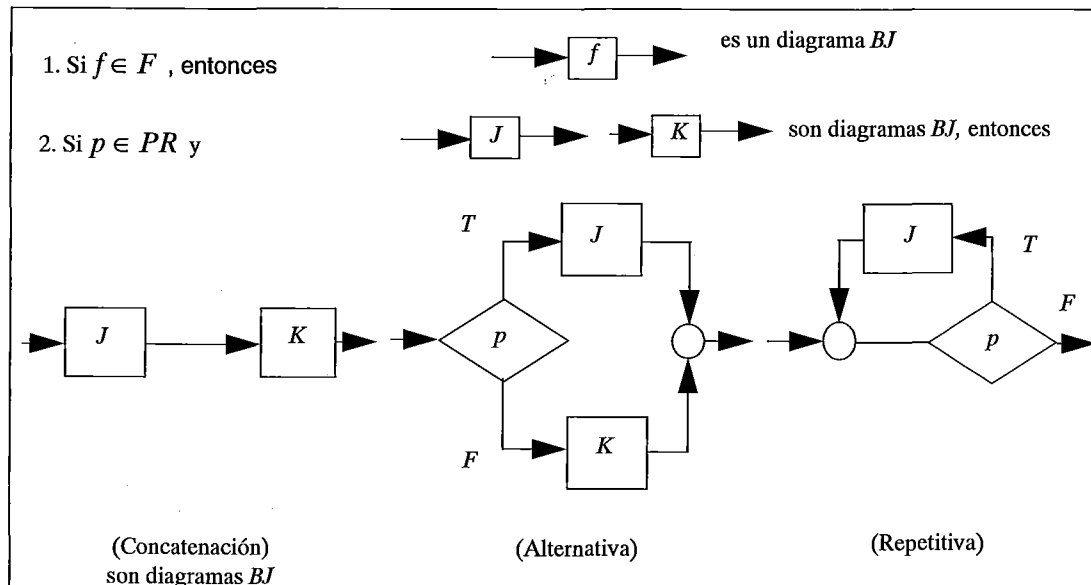


Figura 3.3.

3. ¿Qué es un programa estructurado?

3.1. Estructuras básicas

Se ha demostrado que toda función de programa puede realizarse mediante un programa limpio construido mediante diagramas BJ (llamados a veces diagramas D, por Dijkstra, uno de los autores más reconocidos). Algo semejante ocurre, como vimos (Tema "Lógica", cap. 2, apartado 2.4), con las expresiones lógicas del cálculo proposicional, que pueden construirse a partir de sólo dos operadores lógicos: OR y NOT.

Sin embargo, en la práctica suelen utilizarse dos bloques básicos más, derivados de los anteriores, con objeto de simplificar los programas. La figura 3.4 presenta los cinco bloques básicos que vamos a utilizar a lo largo de este capítulo.

3.2. Definición de programa estructurado

Se dice que un programa es estructurado si se expresa únicamente mediante las estructuras básicas de la figura 3.4. En concreto, las funciones de programa de cada una de las estructuras anteriores son:

$$\text{BLOCK}(f,g,\dots,h)(q) = h(\dots(g(f(q)))).$$

$$\text{IFTHENELSE}(p,f,g)(q) = \text{Si } p, f(q), \text{ si } \neg p, g(q).$$

GRAFO	FORMULA	Expresión	
		Pascal	C
1.	BLOCK(f,g)	begin f;g end	{f;g;}
2.	IFTHENELSE (p,f,g)	if p then f else g;	if (p) then f else g;
3.	DOWHILE(p,f)	while p do f;	while (p) f;
4.	IFTHEN(p,f)	if p then f;	if (p) then f;
5.	DOUNTIL(p,f)	repeat f until p;	do f while(p);

Figura 3.4.

$IFTHEN(p,f)(q) = Si\ p, f(q),\ si\ \neg p, q.$

$DOWHILE(p,f)(q) = Si\ p, DOWHILE(p,f)(f(q)),\ si\ \neg p, q.$

$DOUNTIL(p,f)(q) = Si\ p, DOUNTIL(p,f)(f(q)),\ si\ \neg p, f(q).$

donde q es un estado cualquiera de la máquina. Se observará que hemos extendido la estructura básica BLOCK para que acepte más de dos argumentos. En general, esta extensión puede expresarse así (en el caso de tres argumentos):

$BLOCK(f,g,h) = BLOCK(f,BLOCK(g,h))$

Un programa será estructurado si su función puede expresarse de una de las cinco formas anteriores, en función de cierto conjunto de predicados p, y donde cada una de las funciones f, g, es:

La función identidad.

Cálculos de P.

Un subprograma limpio de P, también estructurado.

Ejemplo: el grafo de la figura 3.5 es estructurado porque su función puede escribirse así:

$[P] = IFTHENELSE\ (p, DOWHILE\ (q, f), BLOCK\ (g, h))$

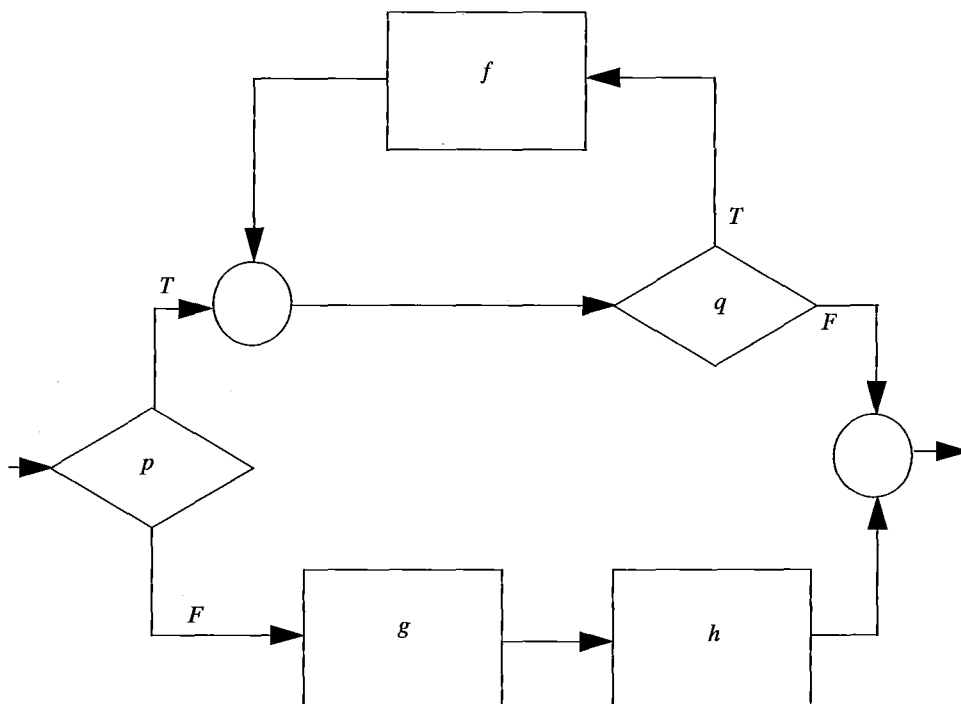


Figura 3.5.

3.3. Teorema de estructura para un programa limpio

Existe un teorema de estructura que, dado un programa P (representado por un diagrama de flujo cualquiera), demuestra la existencia de un diagrama BJ equivalente a P (en el sentido de representar la misma función de programa $[P]$), que utiliza el mismo conjunto de funciones básicas y de predicados asociados a P , junto con tres funciones nuevas y un predicado adicional.

Las funciones nuevas son:

- TRUE: Añade el valor "true" (cierto) al conjunto de sus datos de entrada. Es decir, $\text{TRUE}(a) = (a, \text{true})$, donde a son los datos de entrada del bloque asociado a esta función.
- FALSE: Añade el valor "false" (falso) al conjunto de sus datos de entrada. $\text{FALSE}(a) = (a, \text{false})$.
- POP: Es la función inversa de las dos anteriores. Elimina el último valor lógico añadido a su conjunto de datos de entrada. Es decir, $\text{POP}(a,b) = a$, donde b es un valor lógico ("true" o "false").

El predicado nuevo es:

TOP: Aplicado a un conjunto de datos al que se ha añadido un valor lógico, su resultado es dicho valor lógico. Es decir, $TOP(a,b) = b$, donde b es un valor lógico ("true" o "false").

El teorema de estructura, que no vamos a demostrar, dice así: todo *programa limpio* es equivalente (en el sentido de representar la misma función de programa [P]) a un programa estructurado que contiene como máximo las fórmulas BLOCK, IFTHENELSE (IFTHEN), y DOWHILE (o DOUNTIL), las funciones TRUE, FALSE y POP, el predicado TOP, así como las funciones y predicados del programa original.

3.4. Ejemplo de aplicación del teorema de estructura

Partimos del organigrama de la figura 3.6.

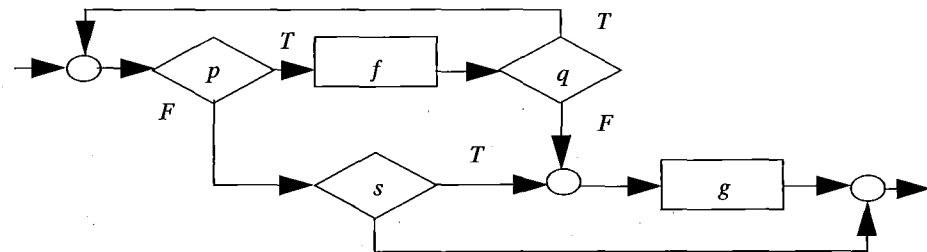


Figura 3.6.

Aplicando el teorema de estructura, podemos construir el de la figura 3.7,

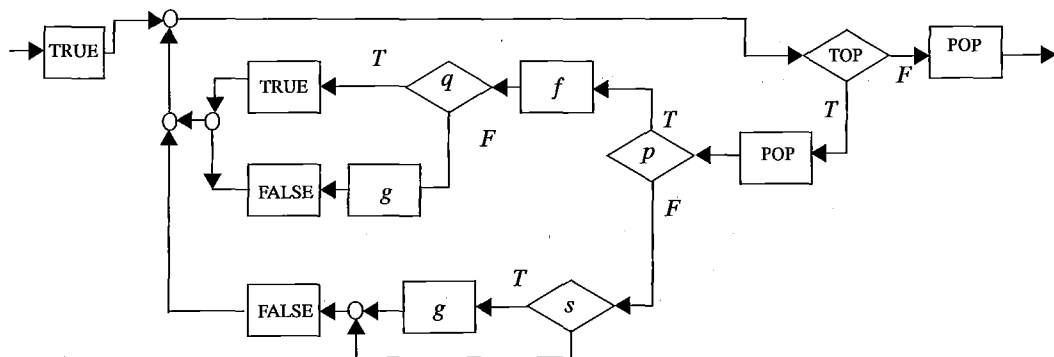


Figura 3.7.

equivalente a él. Obsérvese que al comienzo de la ejecución se realiza la secuencia de sentencias (TRUE;TOP;POP;p), por lo que la función p tiene como dominio el mismo conjunto de datos de partida que el programa original. En el ramal donde se encuentra la función f marcamos uno de los arcos con TRUE y el otro con FALSE para poder encaminar de nuevo la ejecución hacia el predicado p o hacia la salida del programa. El ramal inferior (el de la secuencia (s;g) se marca con FALSE para abandonar la ejecución la próxima vez que se pase por el predicado TOP.

Este organigrama puede describirse mediante la siguiente fórmula o función de programa:

$$[P] = \text{BLOCK} (\text{TRUE}, \text{DOWHILE} (\text{TOP}, \text{BLOCK} (\text{POP}, \text{IFTHENELSE} (\text{p}, \text{BLOCK} (\text{f}, \text{IFTHENELSE} (\text{q}, \text{TRUE}, \text{BLOCK} (\text{g}, \text{FALSE}))), \text{BLOCK} (\text{IFTHEN} (\text{s}, \text{g}), \text{FALSE})))), \text{POP}).$$

4. Método general de diseño de programas estructurados

Del teorema anterior se deduce, como corolario, que todo programa estructurado se reduce a una sola de las estructuras básicas: BLOCK, IFTHENELSE (IFTHEN), DOWHILE (o DOUNTIL). El ejemplo anterior se reduce a BLOCK. El método de diseño de programas estructurados parte de ahí: su primer objetivo debe ser deducir cuál es la estructura básica primordial del programa que se desea construir. A partir de ese punto se irá profundizando en el interior de esa estructura, refinándola progresivamente e introduciendo niveles y pasos sucesivos ("stepwise refinement") hasta llegar a un nivel en el que ya no interese refinar más.

4.1. Recursos abstractos

Para resolver un problema por los métodos clásicos de la programación procedimental, es preciso pasar por cinco etapas fundamentales (solución en cascada):

- Análisis del problema (qué hay que hacer).
- Diseño de una solución (cómo lo hacemos).
- Codificación del programa (hagamos que lo haga).
- Pruebas del programa (comprobemos que lo hace).
- Mantenimiento (si no lo hace perfectamente bien).

En general, existe una brecha importante entre las especificaciones de un problema a resolver (resultado del análisis) y los recursos concretos de que se dispone

(bloques básicos del diseño), brecha que es necesario salvar. Uno de las maneras posibles para resolver este problema (veremos otra posibilidad en el capítulo siguiente) consiste en descomponer las acciones complejas en otras acciones más simples (*recursos abstractos*, según Dijkstra) concebidas como instrucciones para una supermáquina (inexistente) capaz de poder ejecutarlas. Como es natural, una vez que hayamos descompuesto el problema de esta manera, tendremos que procesar esas superinstrucciones, que se convertirán de esta forma en subobjetivos, y para ello podemos aplicar recursivamente el mismo método. Esta manera de concebir los programas no es exclusiva de la programación estructurada, pues se aplica igualmente en ciertas formas de la programación modular. En nuestro caso, sin embargo, hay que recordar que la descomposición en superinstrucciones debe corresponderse con las estructuras básicas de las que hemos hablado en apartados precedentes. Por lo tanto, el razonamiento utilizado para el diseño debe guiarse por la decisión de utilizar siempre esas estructuras básicas en cada uno de los pasos sucesivos, como veremos a continuación.

4.2. Razonamiento deductivo (diseño descendente)

Con este nombre se designa al proceso mental que permite construir un programa por medio de una marcha analítica que se refleja en niveles o pasos sucesivos de refinamiento, cada uno de los cuales posee sus propios recursos abstractos que permiten resolver el programa por completo. Esta marcha analítica, totalmente cartesiana, exige que en cada nivel de razonamiento podamos apoyarnos en el hecho de que el problema ha sido totalmente resuelto para el nivel superior. Sólo se trata de construir un refinamiento que se acerque más a las limitaciones existentes (los recursos reales). Por eso *nunca será necesario volver atrás*, salvo por haber detectado un error en la fase anterior o por tratar de encontrar una solución más conveniente.

El paso de un nivel al nivel siguiente se realiza mediante un cambio en el "punto de vista", lo que permite introducir un refinamiento. Obsérvese que cualquiera de nuestras estructuras básicas tiene sólo un punto de entrada y una salida única, como indica la figura 3.8. Podemos ver cada una de ellas de dos maneras diferentes:

Punto de vista "exterior". La estructura es una caja negra en la que penetra la información de entrada, para ser manipulada y transformada en la información de salida correspondiente. En este caso interesa sólo *lo que hace* la estructura, pero no cómo lo hace.

Punto de vista "interior". La situación es la opuesta. Ahora nos interesa *cómo lo hace*, encadenando acciones, ofreciendo alternativas o repitiendo acciones más simples. En este momento, lo que nos interesa es averiguar la estructura interna de la caja negra primitiva.

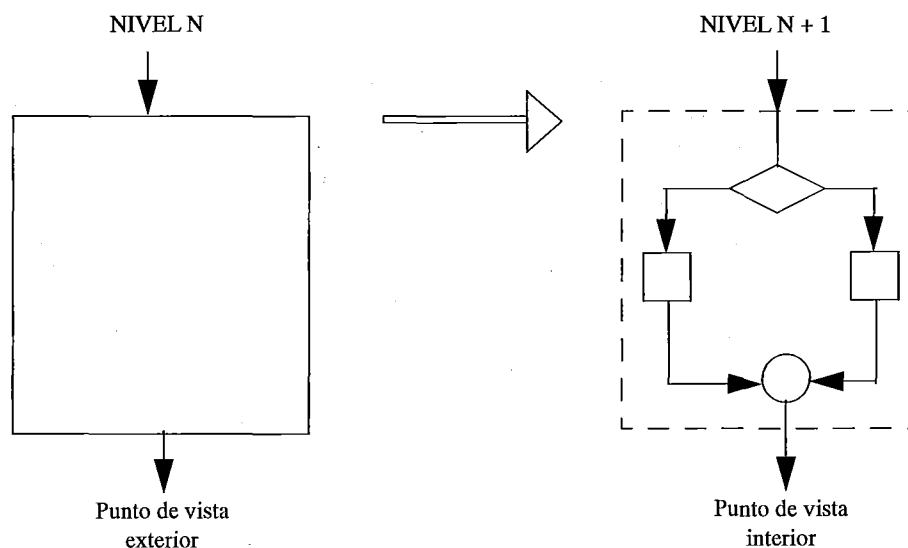


Figura 3.8.

4.3. Ejemplo de diseño de un programa estructurado

Se ha escogido un caso práctico que consiste en simular un reloj digital y estructurar su programación. El reloj tendrá dos ventanillas visualizadoras, la primera con tres campos (horas, minutos y segundos), cada uno de dos dígitos; la segunda con dos campos (mes y día), también de dos dígitos. Por ejemplo, el 30 de abril se representará en la segunda ventanilla como 04:30.

Para que la simulación sea completa, dotaremos a este reloj de una batería que se agota al cabo de N impulsos, siendo N un dato numérico conocido. Por lo tanto, los datos de entrada de nuestro programa serán seis: N , HORA, MINUTO, SEGUNDO, MES y DIA, donde los cinco últimos definen el momento inicial, en que el reloj se pone en marcha.

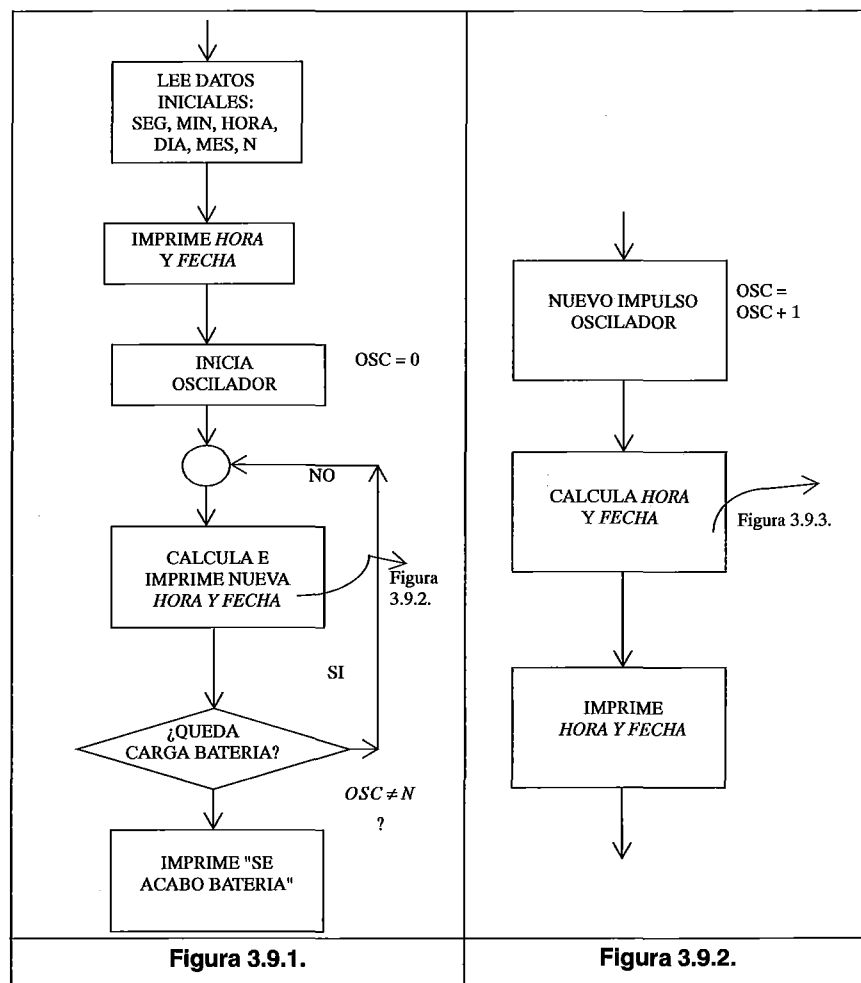
Se desea que el programa escriba sucesivamente "HORA", hora actual, "MINUTO", minuto actual, "SEGUNDO", segundo actual, "DIA", día actual, "MES", mes actual, una línea de seis asteriscos, "HORA", hora actual, etc. Cuando se acabe la batería, el programa escribirá "SE ACABO LA BATERIA" y se detendrá. Los caracteres entrecomillados denotan textos que hay que imprimir literalmente.

Recordemos que febrero *tiene 28 días* (se desprecia la consideración de los años bisiestos), abril, junio, septiembre y noviembre tienen 30, el resto de los meses 31. Obsérvese también que el programa debe escribir los estados sucesivos de la esfera del reloj, pero no le exigimos que lo haga a intervalos de tiempo fijo.

En el primer nivel planteamos una solución esquemática completa que resuelve totalmente el problema, si fuera posible contar con una máquina que sea

capaz de ejecutar las instrucciones contenidas en los distintos nodos del organigrama de la figura 3.9.1. En este esquema, las palabras *HORA* y *FECHA*, escritas en cursiva, representan los conjuntos (*HORA*, *MIN*, *SEG*) y (*MES*, *DIA*), respectivamente. Cuando el texto *HORA* no aparezca en cursiva se refiere al elemento *HORA*.

Obsérvese que se utiliza un contador, una variable interna o local, que llamaremos *OSC* (de *OSCILADOR*), porque corresponde al oscilador de un reloj digital real. Si se exceptúa la instrucción "CALCULA E IMPRIME NUEVA *HORA* Y *FECHA*", las demás instrucciones de la figura 3.9.1 son traducibles de forma sencilla al lenguaje de cualquier ordenador. Por ello prescindiremos de ellas y nos dedicaremos con más detalle a la instrucción mencionada, que desglosaremos en un nivel superior (figura 3.9.2).



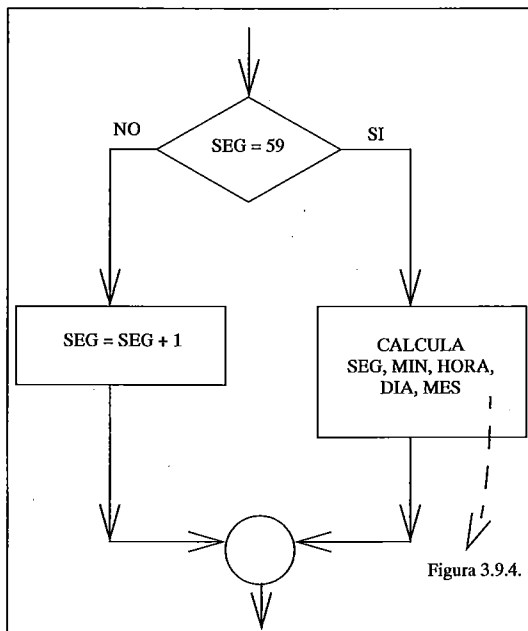


Figura 3.9.3.

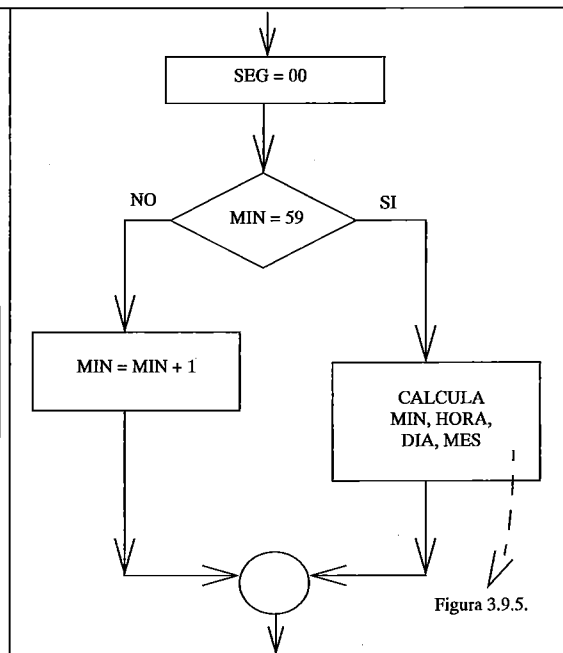


Figura 3.9.4.

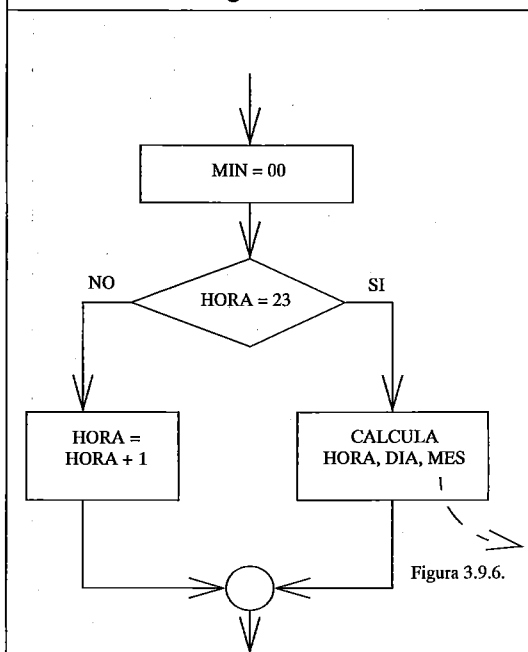


Figura 3.9.5.

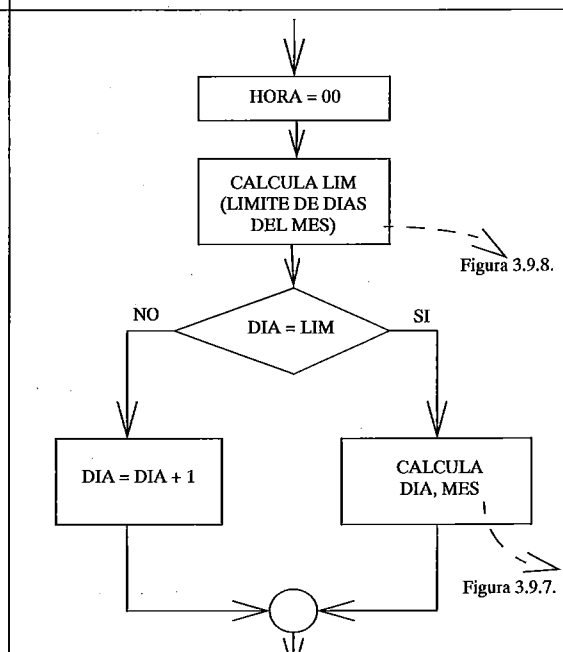
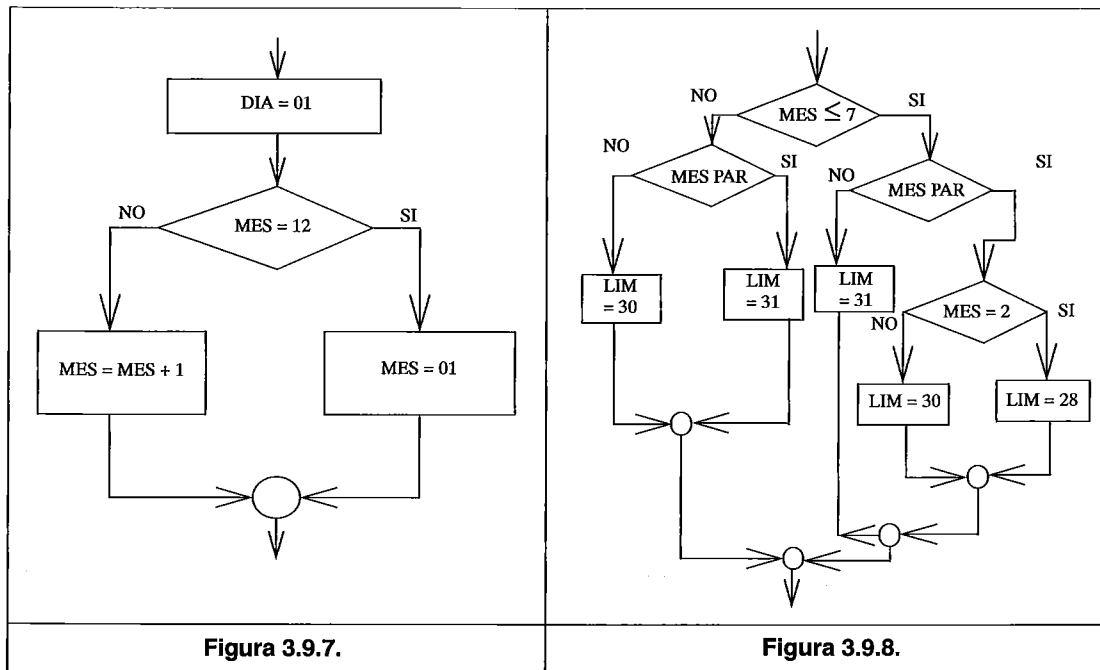


Figura 3.9.6.



El refinamiento continúa progresivamente a través de niveles sucesivos representados por las figuras 3.9.3, 3.9.4 y 3.9.5. Al llegar al nivel representado en la figura 3.9.6 nos encontramos ante un pequeño obstáculo: todos los meses no tienen el mismo número de días. Lo que hacemos es recurrir a un recurso abstracto que calcule el límite de la cuenta de días en un mes, dejando su desarrollo para más adelante, en este caso para la figura 3.9.8, donde se plantea una de las soluciones estructuradas posibles.

Una vez terminado el proceso de desglose, se recorre el camino en sentido inverso sustituyendo sistemáticamente los diagramas más sencillos en el diagrama de nivel inmediatamente anterior, hasta recuperar la estructura completa de la figura 3.9.1, pero ya compuesta con todo el detalle de las operaciones fácilmente traducibles a instrucciones para un ordenador. El organigrama final es el de la figura 3.10 donde, por razones de espacio, no se ha sustituido el diagrama de la figura 3.9.8. Es obvio que este diagrama dista de ser el mejor. Por ejemplo, sería posible introducir una variable adicional, a la que se asignara el número de días del mes cada vez que éste se inicie.

5. Programación estructurada en Pascal y C

Pascal y C son dos lenguajes especialmente diseñados para la programación estructurada. De hecho, la instrucción GOTO está francamente desaconsejada, y algunos compiladores incluso generan un mensaje de aviso cuando se encuentran una en el programa fuente.

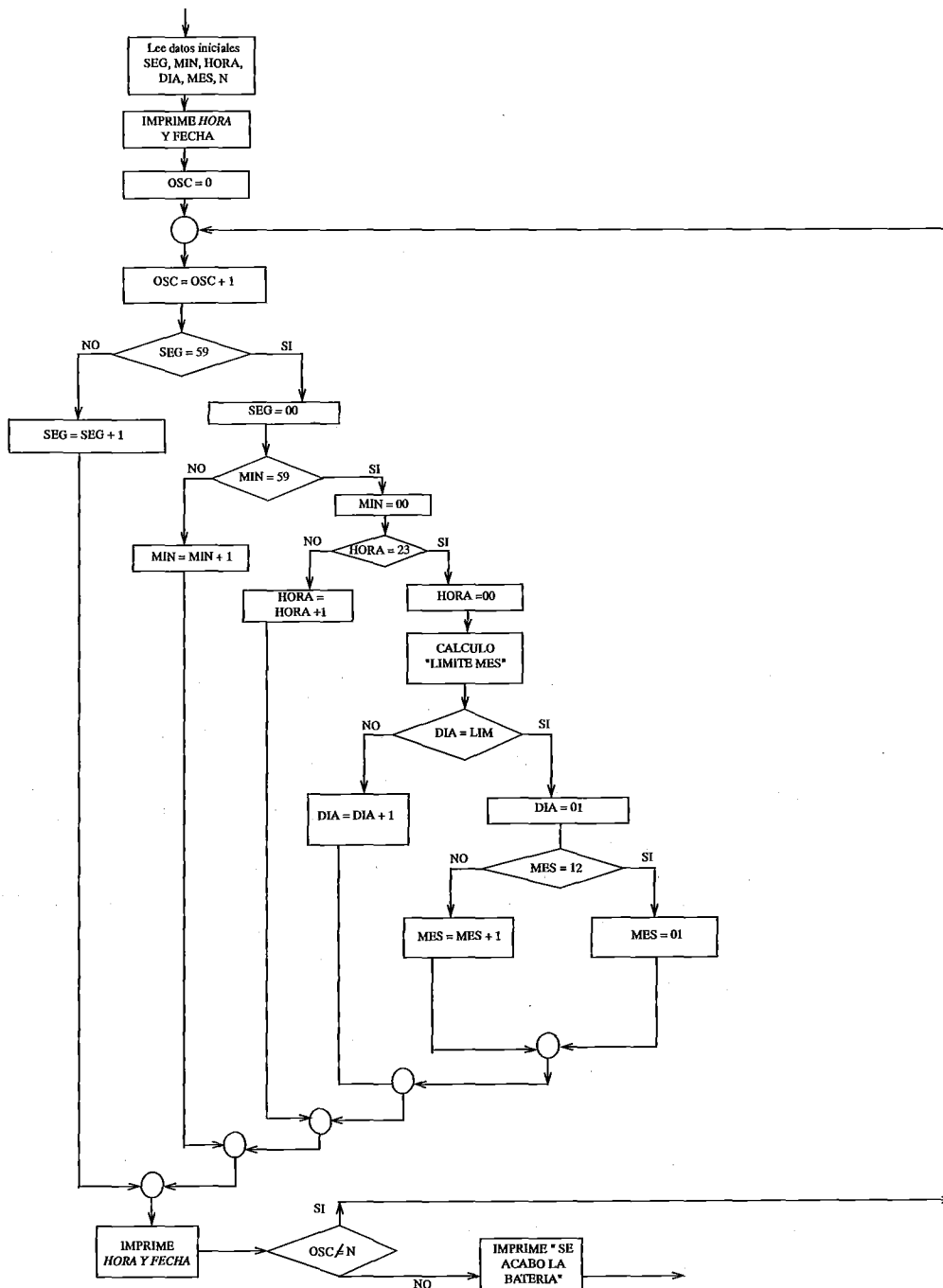


Figura 3.10.

Además de las estructuras básicas, ya detalladas en la figura 3.4, ambos lenguajes permiten utilizar otras, que definiremos a continuación.

5.1. Bloque múltiple

Como hemos dicho más arriba, se suele extender la definición de la estructura BLOCK para que puedan concatenarse más de dos funciones. La extensión correspondiente en Pascal y en C es elemental. BLOCK(f,g,...,h) se representa así:

En Pascal:

```
begin
  f;g;...;h
end
```

En C:

```
{f;g;...;h}
```

5.2. Bucle "for"

Una de las formas más corrientes de la instrucción en bucle utiliza un contador que va aumentando o disminuyendo de valor hasta alcanzar cierto valor de terminación, en cuyo caso el bucle deja de ejecutarse. Esta instrucción es fácilmente reducible a un bloque DOWHILE, por lo que su introducción desempeña el papel de una simple abreviatura. Veamos un ejemplo en cada uno de los dos lenguajes:

En Pascal:

```
for N:=1 to 10 do f;
```

En C:

```
for (N=1; N<=10; N++) f;
```

El ejemplo anterior es equivalente a:

```
BLOCK (N=1, DOWHILE (N<=10, BLOCK (f, N++)))
```

5.3. Instrucción CASE

Esta es otra instrucción que abrevia varias de las clásicas en una sola. En particular, tiene sentido utilizarla cuando tenemos que introducir en nuestro progra-

ma cierto número de IFTHENELSE consecutivos cuyos predicados respectivos son comparaciones de valor de una variable con dos o más constantes. Veamos un ejemplo en Pascal y C:

En Pascal:

```
case N of
  1: X:=3;
  3: X:=7;
  5: X:=2;
  else X:=5
end
```

En C:

```
switch (N) {
  case 1: X=3; break;
  case 3: X=7; break;
  case 5: X=2; break;
  default: X=5; break;
}
```

El ejemplo anterior es equivalente a:

```
IFTHENELSE (N==1, X=3,
  IFTHENELSE (N==3, X=7,
    IFTHENELSE (N==5, X=2, X=5)))
```

6. Ventajas de la programación estructurada

La programación estructurada (P.E.) se impuso durante los años setenta porque proporcionaba a los programadores ventajas evidentes importantísimas, entre las que podemos destacar las siguientes:

Comunicabilidad, considerada como una faceta dentro de la condición más general de *inteligibilidad*. La P.E. *produce programas claros, limpios, expresados con fórmulas y diagramas muy fáciles de comprender*. La consecuencia inmediata es la disminución de los costes de programación y la potenciación del trabajo en equipo. También es importante la faceta didáctica: la P.E. puede enseñarse bien, ya que facilita la discusión y la comparación de programas.

Corrección: el aspecto más importante de la programación estructurada es *la posibilidad de demostrar formalmente si un programa es correcto o no*. En

efecto, si la fórmula de un programa emplea sólo las estructuras básicas, puede probarse si es correcto o no mediante un censo de todos los nodos del grafo. Dejamos sugerida esta cuestión, de gran complejidad teórica, que se sale del objetivo de este libro.

Facilidad de modificación: el diseño multinivel que hemos detallado en las páginas anteriores abre paso a la posibilidad de introducir modificaciones más o menos importantes en una aplicación previamente construida. Imagine el lector, por ejemplo, la posibilidad de extender el ejemplo del reloj digital, introduciéndole una tercera ventana donde aparezca el día de la semana.

7. Resumen

Se ha definido un programa como la *representación de un algoritmo*. Análogamente, un *organigrama* (u *ordinograma* o *diagrama de flujo*) es un grafo que *representa un programa*.

Se ha visto que el organigrama de cualquier programa *puede construirse con sólo tres tipos de nodos: de función, de predicado y de agrupamiento*. Además, cualquier programa puede sustituirse por otro equivalente (en cuanto a la función que realiza) cuyo organigrama *está construido con sólo tres tipos de estructuras o diagramas simples: concatenación, alternativa y repetitiva*. Estas tres estructuras simples tienen una entrada y una salida única. Un programa está estructurado cuando su organigrama está formado únicamente por las tres estructuras simples, o por otras, igualmente simples, derivadas de ellas.

Si un programa es limpio, pero no estructurado, es posible construir un programa estructurado equivalente, sin distorsionar demasiado la estructura original, añadiendo tres funciones (TRUE, FALSE, POP) y un predicado (TOP) que amplían el conjunto de estados introduciendo variables lógicas binarias.

El método general de diseño de programas estructurados se resume en el esquema conceptual de la figura 3.11 sin necesidad de mayores palabras.

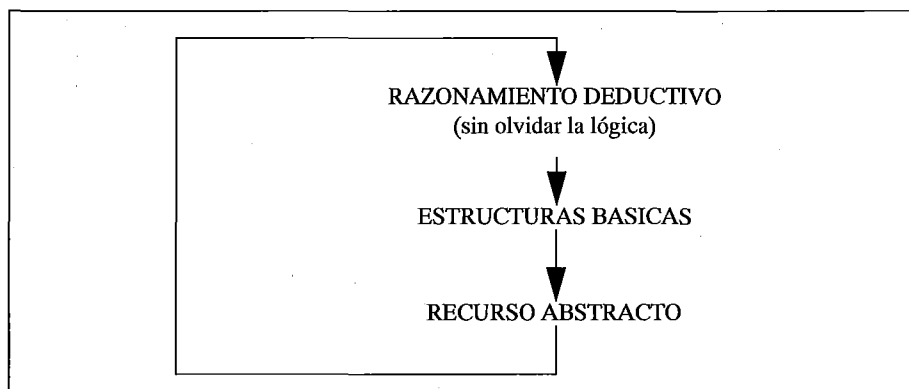


Figura 3.11.

8. Notas histórica y bibliográfica

En 1966, Böhm y Jacopini pusieron las primeras piedras de la teoría de la programación estructurada, partiendo de los diagramas o estructuras básicas indispensables en un lenguaje de programación. Otras aportaciones teóricas en este campo se deben a Dijkstra, Hoare, Wirth y Mills (Mills, 1975). Es notable el libro de Dahl, Dijkstra y Hoare (1972). Otras obras interesantes corresponden a Manna (1974), Dijkstra (1976) y Alagic y Arbib (1978).

Más al alcance de los programadores profesionales, es de notar el impacto causado por un número de la revista *Datamation* (1973), que dedicó varios artículos a este tema con el título genérico de "Revolución en la Programación".

No se pretende que estas referencias constituyan siquiera un conjunto mínimo básico. Sólo estudiando una obra colectiva preparada por distintos especialistas puede uno hacerse una idea de conjunto. Merece la pena citar un estudio panorámico en español, editado por Gamella (1985).

En este capítulo, el teorema de estructura del subapartado 3.3 se tomó de Tabourier et al. (1975), referencia utilizada asimismo en varios pasajes del capítulo.

En cuanto al método de diseño de programas estructurados, se debe a Dijkstra (en Dahl *et al.*, 1972). Se trata de un método muy general, utilizable en toda circunstancia. Sin embargo, motivos prácticos han aconsejado el desarrollo de procedimientos dirigidos a cierta clase de problemas, como los de Yourdon, Linger y Mills (EE.UU., véase Yourdon, 1975), Jackson (G.B., 1975), Warnier y Bertini (véase Warnier, 1973). Sáez Vacas (1976) ha estudiado algunos de estos métodos y su relación con los fundamentos científicos de la P.E.

El ejemplo de simulación de un reloj digital lo hemos adaptado de Arbib (1977).

Para ampliar la programación estructurada en Pascal o en C pueden utilizarse los dos textos clásicos de los autores de estos lenguajes: Kernighan y Ritchie (1978) y Wirth (1980), o bien libros más modernos, como Kelley y Pohl (1987), Gottfried (1993), Joyanes (1990) o Salmon (1993). Sobre programación en general, puede verse Cerrada y Collado (1993), que emplea el lenguaje Modula-2.

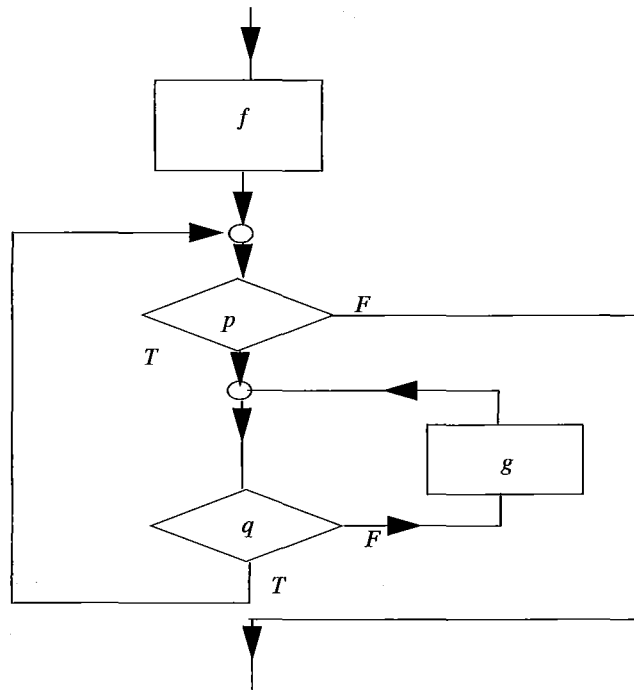
9. Ejercicios

9.1. Dado el grafo adjunto, indicar cuál de las siguiente opciones se corresponde con la fórmula de programación estructurada que define los mismos cálculos.

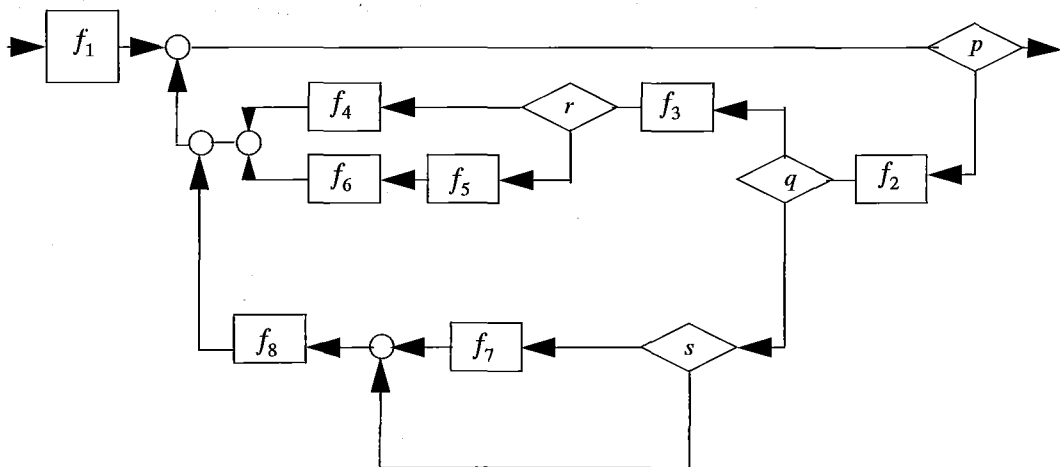
a) BLOCK (f , DOWHILE (p , $\neg q$, g))

b) BLOCK (f , IFTHEN (p , DUNTIL (q , g)))

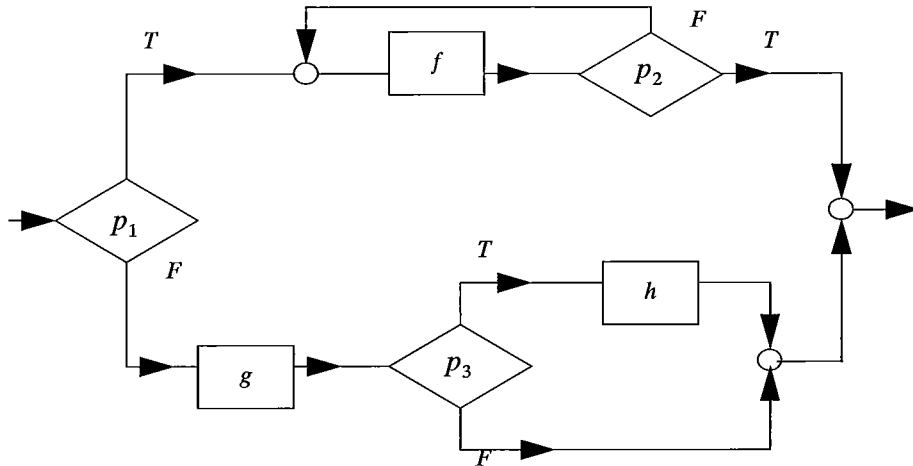
- c) BLOCK (f , DOWHILE (p , DUNTIL (q , g)))
d) BLOCK (f , IFTHEN (p , IFTHEN $\neg q$, g))



9.2. Obtener la fórmula del siguiente programa estructurado.



9.3. Obtener la fórmula correspondiente al siguiente grafo estructurado:



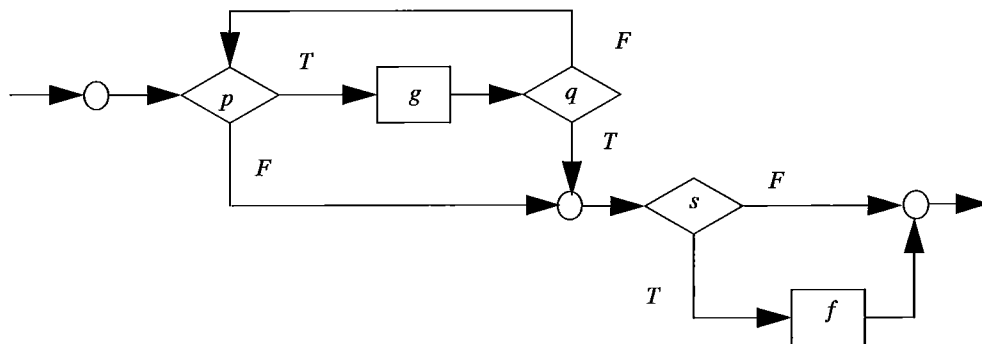
9.4. Obtener el organigrama del siguiente problema estructurado:

DOWHILE (p_1 , IFTHENELSE (p_2 , IFTHEN (p_3 , BLOCK (DO-UNTIL (p_4 , DOWHILE (p_5 , f_1)), IFTHENELSE (p_6 , f_2 , f_3))))))

9.5. Obtener el diagrama de flujos del programa estructurado siguiente:

BLOCK (f_1 , DOWHILE (p_1 , IFTHENELSE (p_2 , f_2 , IFTHENELSE (p_3 , IFTHENELSE (p_4 , f_3 , f_4), IFTHENELSE (p_5 , f_5 , IFTHEN (p_6 , f_6))))))

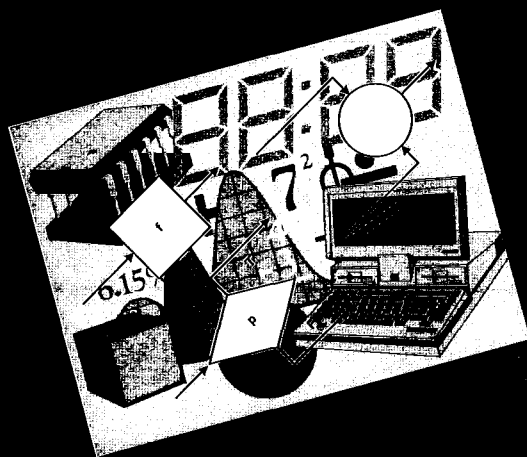
9.6. Transformar el siguiente organigrama en un organigrama estructurado utilizando el método de las cuatro funciones estándar TRUE, FALSE, POP y TOP. Nota: pivótese básicamente sobre la estructura DOUNTIL.



Escríbase la fórmula de la estructura resultante.

Fundamentos de informática

4



4

Evolución de la programación

1. Introducción

En este capítulo profundizamos en las metodologías de análisis y diseño apuntadas en el capítulo anterior al hablar de recursos abstractos. Se discute también cómo se puede asegurar que el programa desarrollado para plasmar un algoritmo es correcto (no contiene fallos) o, en caso de contenerlos, es robusto (los detecta y actúa ante ellos de forma ordenada).

Actualmente, la orientación a objetos se ha convertido en una de las metodologías de análisis, diseño y programación más extendidas. Por ello, su descripción somera no podía faltar en este libro, aunque no sea éste el lugar adecuado para entrar a fondo en la cuestión. Para compensar un tratamiento tan sucinto, el capítulo termina con unas "Notas" relativamente amplias, dirigidas a los lectores más interesados.

2. Cómo se construye un programa

A principios de los años ochenta se llegó a establecer una metodología, que podríamos llamar estándar, para el desarrollo de aplicaciones de programación. Esta metodología, que no es más que una generalización, aplicada a la informática, de la que mencionamos al referirnos a la resolución de problemas en el capítulo 1 (figura 1.1), tiene el aspecto que indica la figura 4.1.

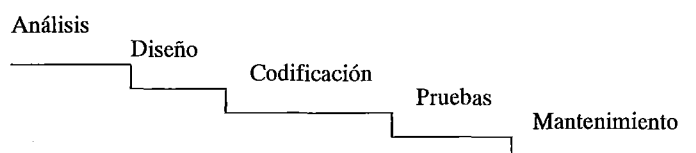


Figura 4.1.

Debido al aspecto de la figura, se dice que esta metodología se desarrolla en *cascada*, pues parte de la base de que no se puede pasar a una de las cinco etapas sin haber dado por terminadas todas las anteriores.

Las etapas en cuestión pueden definirse así:

- **Análisis:** equivale a decidir *qué es lo que tenemos que hacer*. Es decir, definir la aplicación o programa a construir, en función de los requisitos establecidos por el usuario. Llamamos usuario a la persona que encarga el programa, que no tiene necesariamente por qué coincidir con la persona que va a utilizarlo. Para distinguirlos, a este último lo llamaremos *usuario final*.
- **Diseño:** en esta etapa, lo que hay que definir es *cómo vamos a hacerlo*. Entre todas las formas posibles de resolver el problema, se trata de elegir la más simple, o la más barata, o la más eficiente, o la que mejor se adapte a los requisitos del análisis.
- **Codificación:** una vez sabemos lo que hay que hacer, y cómo hacerlo, ha llegado el momento de la verdad: hay que ponerse a programar y construir la aplicación. Esta es, normalmente, la parte más larga del proceso y la que consume más recursos.
Pruebas: no basta con que el programa esté terminado, hay que asegurarse de que funciona perfectamente, no sólo en un caso, sino en todos. Un buen diseño de pruebas es esencial para esto. Otra alternativa podría ser la comprobación automática de programas. (Tema "Lenguajes", cap. 5, apartado 4.1)
- **Mantenimiento:** a pesar de todos nuestros esfuerzos, si la aplicación es lo suficientemente grande o compleja, es casi seguro que en la fase de pruebas se nos escapará alguna situación en la que el programa no funcione correctamente. Por eso hay que tener previsto que tendremos que volver a él para

corregir los problemas que puedan ir surgiendo, y para ello es esencial que el programa esté muy bien documentado. Además, puede que sea necesario volver de nuevo a las fases de análisis y diseño, para corregir, mejorar o adecuar la aplicación, ya sea porque vamos a construir una segunda versión, o simplemente para responder a cambios en las condiciones de contorno de la aplicación a lo largo de su vida útil.

En la metodología en cascada, es preciso dar por terminado el análisis antes de comenzar con el diseño; hasta que no terminemos con éste no empezaremos a programar; no haremos las pruebas hasta que todo el programa esté completo; y no comienza la fase de mantenimiento hasta que demos por terminadas las pruebas.

3. Técnicas de análisis y diseño

La metodología en cascada puede ampliarse, subdividiendo las cinco etapas indicadas y aumentando el grado de detalle dedicado a cada una de ellas. Por ejemplo, existe una metodología llamada MAPS (siglas de *Methodology for Algorithmic Problem Solving*), cuyas siete etapas (véase la figura 4.2) pueden reducirse a la metodología en cascada de la siguiente manera:

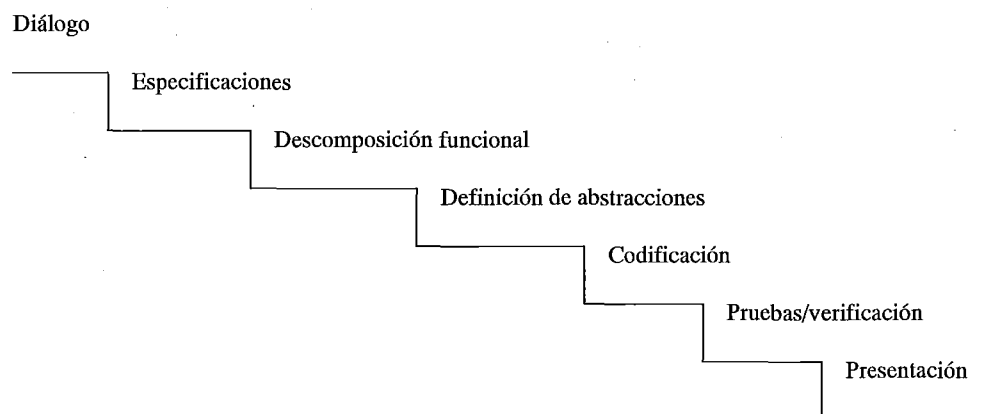


Figura 4.2.

- Etapa 1: diálogo. Es esencialmente idéntica a la fase de análisis.
- Etapa 2: especificaciones. Es la primera subfase del diseño, durante la cual se escriben precondiciones, que definen los datos de entrada en todas las formas que pueden tomar, y postcondiciones que especifican el resultado que debe generarse, en función de los datos de entrada, cualesquiera que sean éstos. Las precondiciones y las postcondiciones deben ser consistentes entre sí.

- Etapa 3: descomposición funcional. Es la división sistemática del proceso en una serie de pasos diferentes entre sí. Una vez realizada, la descomposición puede aplicarse de nuevo a cada uno de los pasos obtenidos, reduciendo progresivamente su grado de complejidad hasta que llegue a ser manejable. En esencia, esta etapa ha sido descrita en el capítulo 3, en relación con el diseño descendente de programas estructurados.
- Etapa 4: definición de las abstracciones. Esta es la última subfase del diseño, en la que se decide cómo construir cada una de los pasos obtenidos en la etapa precedente. Este es el lugar adecuado para reutilizar código (si ya hemos tenido que realizar alguna de las operaciones en aplicaciones anteriores) o para decidir si un paso debe construirse como subrutina (si hay posibilidades de usarlo en varios sitios diferentes, ya sea en esta misma aplicación o en otras que puedan venir después), o como código embebido (en caso contrario).
- Etapa 5: codificación.
- Etapa 6: pruebas y verificación. Estas dos etapas son idénticas a la tercera y la cuarta de la metodología en cascada.
- Etapa 7: presentación. Tiene por objeto facilitar el mantenimiento del programa, producto o aplicación construido, añadiendo información sobre quién lo ha construido, en qué fechas, y de qué manera, especificando además qué rutinas de las utilizadas por el programa podrían utilizarse en otros entornos y de qué manera.

La descomposición funcional no es la única metodología de diseño existente. Hay otras, entre las que destacan las siguientes:

El modelo entidad-relación (E-R), que trata de definir las componentes principales de la aplicación y las relaciones que existen entre ellas, por medio de una red semántica (Tema "Lógica", cap. 6, apartado 4) como la de la figura 4.3.

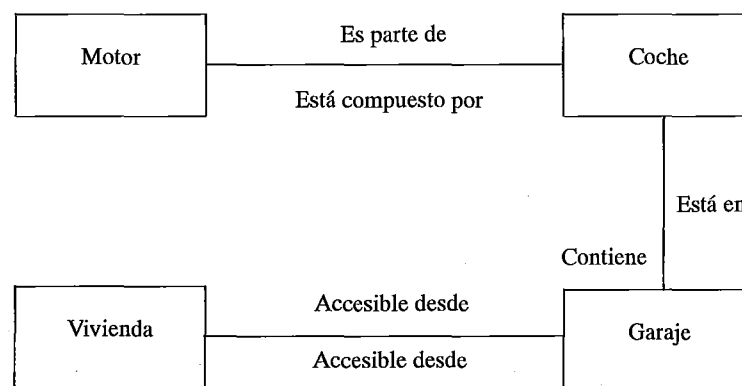


Figura 4.3.

- El diagrama de flujo de datos (DFD), que describe el camino que recorren los datos en un sistema o aplicación, dónde se almacenan, de dónde vienen y adónde van. La figura 4.4 muestra un ejemplo autoexplicativo.

4. Programas correctos, robustos y convivenciales

Uno de los problemas fundamentales cuando se desarrolla un algoritmo y se plasma en un programa de ordenador, es asegurarse de que el programa es, en efecto, idéntico al algoritmo, en el sentido de que ambos producen los mismos resultados para cualquier combinación de datos de entrada. Si esto es cierto, decimos que el programa es *correcto*. Si no lo es, cada discrepancia entre el programa y el algoritmo se considera un *error* del programa. La fase de pruebas (depuración, verificación) de un programa consiste esencialmente en la detección y eliminación de todos los errores.

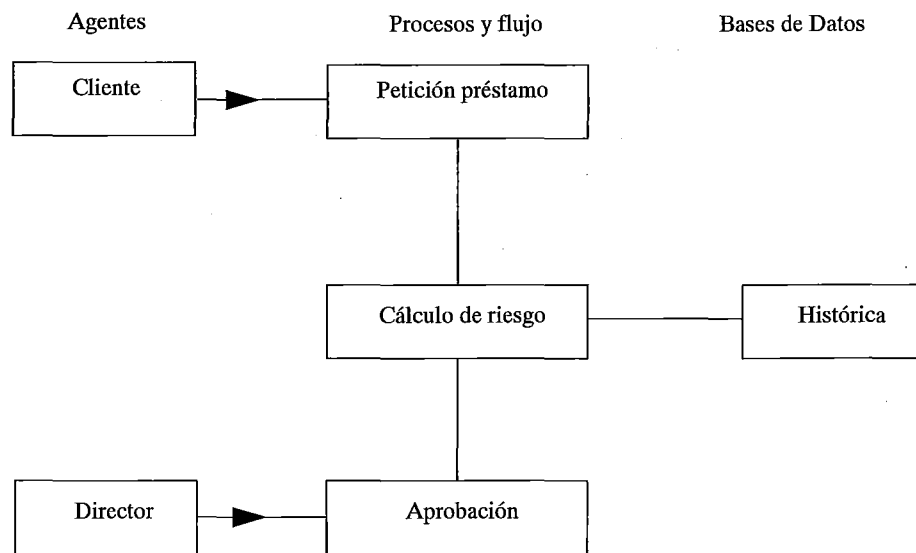


Figura 4.4.

Desgraciadamente, cuando el algoritmo es muy complejo (lo que implica que el programa que lo plasma también lo será) es muy difícil asegurarse de que el programa es correcto. Por eso dijimos, al hablar de la fase de mantenimiento, que es casi seguro que en la fase de pruebas se nos escapará algún error. Todos los fabricantes de software son conscientes de ello y dedican recursos, a veces considerables, al mantenimiento de sus productos.

¿Qué ocurre cuando, a pesar de nuestros esfuerzos, se nos ha escapado algún error? El programa podría reaccionar ante esta situación de maneras muy diferentes:

- Generando un resultado incorrecto, sin avisar de que lo es.
- No terminando nunca. Por ejemplo, metiéndose en un bucle permanente sin salida.
- Terminando de forma desordenada, por ejemplo, saliendo al sistema operativo sin generar mensaje alguno o generando un mensaje muy difícil de interpretar o que no proporcione información útil. Un caso aún peor ocurre cuando el sistema operativo queda inutilizado por culpa del error y es preciso realizar una carga inicial (I.P.L., *Initial Program Load*) o, incluso, volver a encender la máquina.
- Terminando de forma ordenada, generando un mensaje de error significativo que informe al usuario del problema detectado.
- Detectando el error y dando al usuario oportunidad de corregirlo y continuar la ejecución sin tener que volver atrás.

De las cinco posibilidades mencionadas, las tres primeras son erróneas. Quizá la peor de todas sea la primera, pues el programa nos engaña y nos da un resultado incorrecto, del que no tenemos ningún motivo para dudar. La segunda y la tercera son bastante desagradables, pues nos obligan a perder tiempo reiniciando la máquina y el sistema operativo. La figura 4.5 presenta algunos ejemplos de programas de este tipo, escritos en el lenguaje C.

```
/* Algoritmo a realizar: f(a,b)=a+b */
short int suma (short int a, short int b) {
    return a+b;
}
```

Figura 4.5. a)

```
/* Algoritmo a realizar: f(a)=factorial(a) */
float factorial (int a) {
    float b=1.;
    for (;a!=0;a--) b*=a;
    return b;
}
```

Figura 4.5. b)

```

/* Algoritmo a realizar:
   f(a)=longitud de la cadena de caracteres a */
unsigned short longitud (char *a) {
    unsigned short b=0;
    for (;a[0]!='\0';a++) b++;
    return b;
}

```

Figura 4.5. c)

En el programa de la figura 4.5.a), los números a y b han sido definidos como números enteros cortos, usualmente comprendidos entre -32768 y 32767. Si invocamos el programa con los siguientes datos de entrada: suma(20000,30000), no obtendremos la suma correcta (50000), sino un valor erróneo (-15536), debido a que la suma se ha salido del alcance o rango del conjunto de datos con el que estamos tratando (los enteros cortos).

De igual manera, el programa de la figura 4.5.b) se nos mete en un bucle permanente si se invoca con un dato de entrada negativo, por ejemplo, factorial(-1). Y el de la figura 4.5.c) terminará probablemente con un mensaje ininteligible si se invoca con una cadena de caracteres nula, longitud(NULL), pues intentará introducirse en la memoria normalmente reservada para el sistema operativo.

Decimos que un programa es *robusto* si, cualquiera que sea la combinación de sus datos de entrada, no se presenta ninguna de las tres situaciones indicadas. Si, además, en todos los casos se da la situación número 5, diremos que es *convivencial* (*user friendly*, en inglés).

Asegurarse de que un programa es correcto, robusto y convivencial no es nada fácil. Las diversas metodologías de análisis y diseño procuran facilitar el programa descomponiéndolo en partes más simples (como se hace en la fase de descomposición funcional de MAPS o mediante la programación orientada a objetos, que veremos a continuación).

5. Programación orientada a objetos

El estudio de la mente humana ha apasionado y sigue apasionando al hombre desde hace muchos siglos. Poco a poco, vamos obteniendo alguna información sobre su funcionamiento, y es curioso que sea precisamente la gramática, asignatura fundamental en la enseñanza medieval, la que puede arrojar alguna luz sobre el problema de cómo adaptar la programación de los ordenadores al funcionamiento de nuestra mente.

Las técnicas de diseño que hemos descrito en los párrafos anteriores, y que han venido utilizándose tradicionalmente, abordan las aplicaciones desde un punto de vista claramente funcional. Recordemos el procedimiento utilizado en el capítulo

3 para construir un programa que simula un reloj digital. Descompusimos el proceso completo en fases más simples, cada una de las cuales representa una función concreta y viene definida por un verbo de acción: "Lee datos", "Imprime hora", "Inicia oscilador", etcétera. De hecho, el procedimiento de diseño descendente mencionado más arriba (la descomposición funcional) tiene como resultado usual un conjunto de verbos.

Sin embargo, la mente humana evolucionada no funciona así: el verbo no es, normalmente, el elemento primario de nuestra percepción. Hay algo más importante y básico: el nombre sustantivo. Lo primero que nos llama la atención cuando entramos en contacto con el mundo es la multitud de los seres, los objetos. No vemos en primer lugar acciones, sino cosas que actúan o permanecen inertes. Si un insecto pasa volando por nuestro campo de visión, no pensamos "algo vuela", sino "una mosca": el nombre va antes que el verbo, y no sólo sintácticamente. Esto tiene la consecuencia de que nuestras clasificaciones siempre se basan en los nombres, los objetos, nunca en los verbos. Dividimos los animales según sus propiedades físicas (vertebrados o invertebrados, según tengan o no columna vertebral), no según su comportamiento: no se nos ocurre construir el grupo de los que "se arrastran" o de los que "corren".

Entre paréntesis, es sugerente el paralelismo entre la evolución de los modelos de la programación y la del lenguaje humano. Los pueblos primitivos se las arreglan con pocas palabras, casi todos verbos en modo imperativo. Los algonquinos preagrícolas del norte de América poseen ya un vocabulario total de unas 4.500 palabras, con un 53% de verbos y un 32% de nombres. Finalmente, el inglés moderno contiene alrededor de un 10% de verbos sobre un total de unas 150.000 palabras.

Como hemos comentado anteriormente, la programación tradicional se basa principalmente en los verbos, y que éste no es exactamente el método *natural*. ¿No podríamos cambiar de modo de pensar y pasar a una forma de programar basada en los nombres? La respuesta a esta pregunta resulta ser afirmativa. Desde finales de los ochenta ha comenzado a extenderse una metodología alternativa, la programación orientada a objetos (OOP, por las siglas inglesas de *Object-Oriented Programming*). En esta forma de programar, el elemento fundamental no es la acción, sino el objeto, del mismo modo que en las representaciones estructuradas del conocimiento (Tema "Lógica", cap.6 ap. 4). Además, diversos objetos se relacionan entre sí de varias maneras:

Por el hecho de pertenecer a la misma clase o subclase. El conjunto de todas las clases y subclases de objetos con que vamos a trabajar forma una estructura jerárquica como el de la figura 4.6. Los programas clásicos también tenían su jerarquía, pero se basaba en un principio totalmente diferente: en el hecho de que un programa llama a otro (mediante la instrucción CALL o algo equivalente). La jerarquía orientada a objetos queda definida por una relación de pertenencia (un objeto pertenece a una clase) o de subconjunto (una subclase está comprendida en una clase más grande).

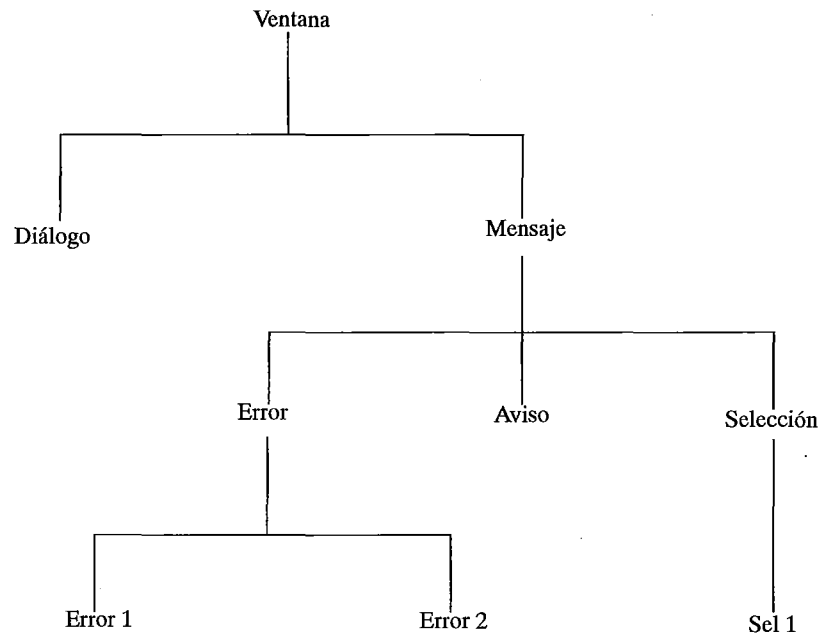


Figura 4.6.

En la figura, Error1, Error2 y Sel1 son objetos, y la relación en que se encuentran con Error y Selección es la pertenencia a una clase. Los restantes arcos del árbol jerárquico representan relaciones de subclasificación (subconjunto).

- Un objeto puede relacionarse con otro de forma no jerárquica, enviándole un mensaje para que haga algo. En la programación orientada a objetos, los mensajes sustituyen a las llamadas de subrutina de la programación clásica. De hecho, la programación orientada a objetos pura se llama a veces *programación sin CALL*, del mismo modo que la programación estructurada también ha recibido el nombre de *programación sin GOTO*.

Dentro de un objeto podemos distinguir, además de sus relaciones con otros objetos, otras dos componentes fundamentales:

- Sus propiedades o atributos (datos asociados al objeto), con sus valores asociados. Del mismo modo que los objetos corresponden a los nombres sustantivos, los valores de sus atributos vienen a desempeñar el papel de los adjetivos. Recuérdese que los números son adjetivos numerales, y que otros valores típicos, como los colores (verde, rojo, etc.) son adjetivos calificativos.

Su comportamiento o funcionalidad (programas asociados al objeto), con su código asociado. En la terminología OOP, los programas se llaman *métodos*. Como hemos dicho, corresponden a los verbos de nuestro vocabulario. Los mensajes OOP dirigidos a un objeto son indicaciones para que ejecute uno de sus métodos. El código de cada método ejecuta o realiza un algoritmo, entendiendo este término en el sentido en que se ha utilizado en los capítulos anteriores. La figura 4.7 resume estos conceptos.

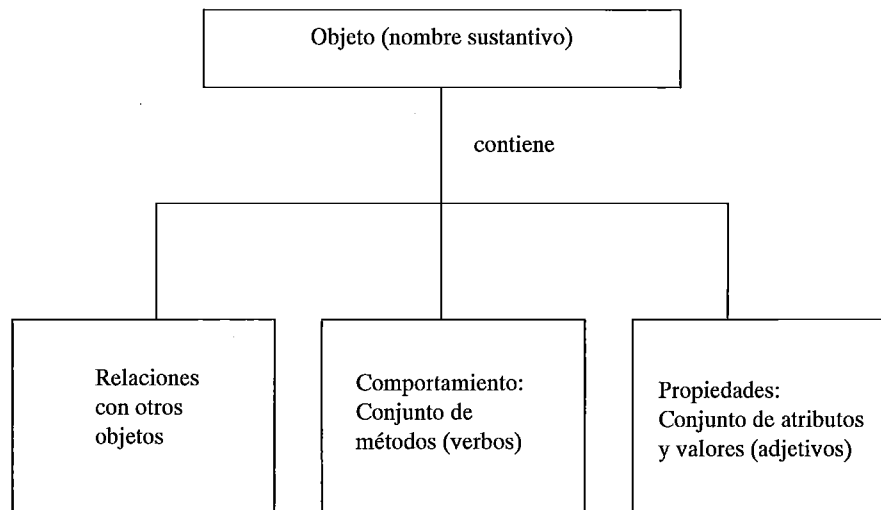


Figura 4.7.

5.1. Propiedades de la programación orientada a objetos

Para que pueda hablarse de programación orientada a objetos no basta con tener objetos y clases. Según P. Wegner, hace falta una componente adicional: la *herencia*, que consiste en el hecho de que los objetos poseen propiedades o métodos, no por sí mismos, sino porque los heredan de las clases a las que pertenecen. Por ejemplo, todos los objetos de la clase "Ventana" tendrán acceso al método "abrir", que estará definido en la clase. De igual manera, la propiedad "tamaño" de la ventana estará definida como tal en la clase "ventana", aunque cada objeto concreto tenga su propio tamaño (un valor diferente para la misma propiedad). Existen dos tipos principales de herencia:

- Herencia simple, si un objeto sólo puede pertenecer a una clase, y cada clase sólo puede ser subclase de una sola superclase.
- Herencia múltiple, si un objeto puede pertenecer a dos clases, o cada clase puede ser subclase de más de una superclase.

Todos los sistemas y lenguajes OOP tienen herencia simple, pero sólo algunos incorporan la múltiple, que por otra parte no es difícil de simular en función de la simple. Además de los conceptos anteriores, hay otros tres que desempeñan un papel fundamental en la programación orientada a objetos:

- **Encapsulamiento:** significa que la información interna de un objeto (los valores de sus atributos) debe estar protegida contra la manipulación directa por parte de otros objetos. Sólo podrá accederse a ella enviándole un mensaje para que ejecute un método. Esta propiedad es muy útil, pues protege a las aplicaciones contra efectos secundarios indeseados, causa frecuente de errores muy difíciles de detectar.

El encapsulamiento nos permite representar los objetos como un huevo de cáscara compacta, que sólo permite el acceso desde el exterior a través de sus relaciones con otros objetos y de su comportamiento (métodos). El interior del huevo se divide en dos secciones (equivalentes a la clara y la yema) y cada una de ellas, a su vez, en otras tres, que son las componentes generales de un objeto que hemos visto en el apartado 5 (véase también la figura 4.8.)



Figura 4.8.

- **Polimorfismo:** es el hecho de que dos objetos distintos pueden tener métodos (programas) que se llamen igual, pero que no se comporten de la misma manera. Por ejemplo: una ventana puede tener un método "abrir", y un fichero también. Los programas no harán lo mismo (no es igual abrir una ventana, es decir, mostrarla en una pantalla, que abrir un fichero, hacer disponible la información que contiene). Un sistema OOP sabrá distinguir cuál de los dos métodos queremos ejecutar, por la clase a la que pertenece el objeto que reciba el mensaje correspondiente.
- **Enlace dinámico:** es una consecuencia del polimorfismo, y significa que el sistema OOP tendrá que diferir a menudo la decisión sobre qué método

concreto debe ejecutarse hasta el instante mismo de la ejecución del programa (no podrá resolverse la cuestión en tiempo de compilación).

5.2. Ventajas de la programación orientada a objetos

Hemos mencionado ya una de sus ventajas fundamentales: la naturalidad del análisis, el hecho de que nuestros modelos mentales se adaptan con más facilidad a una jerarquía basada en clases y nombres que a otra basada en acciones y verbos. Otras ventajas importantes son:

- Es más fácil construir aplicaciones modulares, pues los objetos son módulos naturales: los programas se construyen agrupados por clases de objetos a las que se aplican, y los datos quedan aislados entre sí gracias al encapsulamiento y no interfieren unos con otros.
- Es trivial extender las aplicaciones, pues basta con añadir clases nuevas a la jerarquía, subclasificar las existentes, crear objetos nuevos o dotarles de propiedades y comportamientos adicionales.
- El problema candente de la reutilización del código podría quedar resuelto. Las clases de objetos deberían llegar a convertirse en los bloques básicos que permitirán ensamblar aplicaciones ensamblando unas cuantas piezas.
- Las redundancias quedan eliminadas, pues las propiedades o métodos comunes a varias clases se definirán en una superclase única, siendo heredados por los objetos y clases inferiores a ella en la jerarquía.

Naturalmente, para conseguir estos objetivos es preciso que las clases de objetos estén bien diseñadas. Una clase mal hecha estará demasiado adaptada a una aplicación concreta y no podrá ser utilizada fuera de ella. En OOP es conveniente que las clases se diseñen con vistas a su posible utilización futura en situaciones muy diferentes (generalidad). Esto exige un esfuerzo considerable durante el diseño, que será reintegrado con creces si es posible reutilizarlas con éxito. También es importante que un sistema comercial orientado a objetos facilite al programador bibliotecas de clases prefabricadas bien diseñadas, que faciliten el desarrollo de las aplicaciones, proporcionando un grado considerable de reutilización inicial. En este contexto, se habla a menudo de:

- *"Framework"*: un conjunto de clases que se coordinan para realizar una función. Por ejemplo: interfaces de usuario, aplicaciones bancarias, gestión de bases de datos, etc.
- *Componente*: una clase o pequeño conjunto de clases cuya funcionalidad se puede reutilizar con facilidad, ensamblándola con otras componentes como las piezas de un mecano, para construir una aplicación.

6. Análisis y diseño orientado a objetos

Al construir una aplicación orientada a objetos, utilizamos los mismos conceptos que en la programación clásica, a saber: análisis, diseño, programación, pruebas y mantenimiento. Sin embargo, estas fases se distribuyen de otra manera, y no forman un modelo en cascada, sino un *modelo evolutivo*, que a veces se describe con la frase *diseña un poco, programa un poco, prueba un poco...*

En un entorno orientado a objetos, el análisis consiste en construir un modelo natural de la aplicación, identificando las clases de objetos necesarias para resolver el problema y sus relaciones mutuas (jerárquicas o de otro tipo). Pero con esto no basta: también hay que analizar cada una de las clases en profundidad. El encapsulamiento propio de la orientación a objetos nos asegura que el proceso puede tener lugar de forma incremental, por lo que es posible realizar el diseño, la programación y las pruebas de cada una de las clases independientemente de las demás e, incluso, antes de realizar el análisis de todas ellas, como indica la figura 4.9.

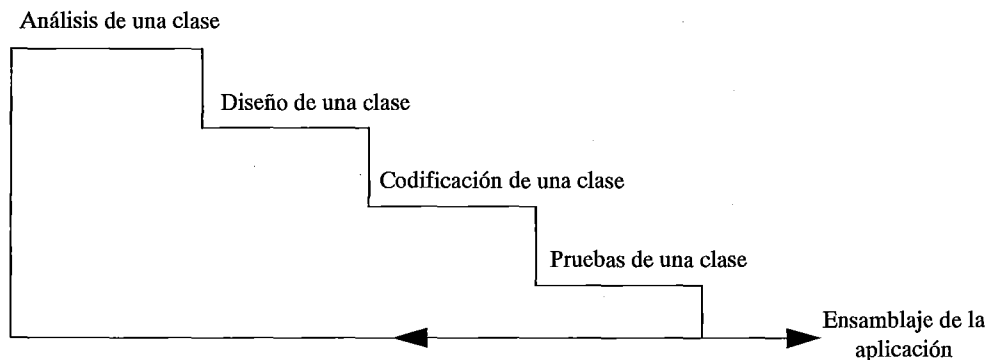


Figura 4.9.

En resumen, el análisis orientado a objetos (OOA) tiene los siguientes objetivos:

- Identificación de las clases de objetos relacionadas con la resolución del problema objeto del análisis.
- Definición de las relaciones entre las distintas clases (a qué superclase pertenecen, en qué subclases se descomponen).
- Especificación de las interacciones entre las diversas clases y objetos (conexiones funcionales, para qué otras clases desempeña el papel de cliente o de servidor).
- Identificación de los atributos de cada clase.
- Identificación del comportamiento de los objetos de cada clase.

El diseño orientado a objetos (OOD) tiene que realizar el diseño en profundidad de cada una de las clases detectadas por el análisis, que incluye el diseño de las estructuras de datos y los métodos (programas) que van a implementar los atributos y el comportamiento de los objetos, así como la identificación de los errores que pueden producirse y la forma de subsanarlos. También le corresponde la identificación y definición de clases de objetos no relacionadas directamente con la resolución del problema, sino con cuestiones de programación, como la interfaz de usuario, la base de datos a utilizar, el soporte de las relaciones entre los diversos objetos, y otras cuestiones básicas y de utilidad.

El diseño orientado a objetos es eminentemente modular, pues ha de conseguir la definición de clases con poco acoplamiento externo, pero con una gran cohesión interna y un mínimo de efectos laterales. Evidentemente, al realizar el diseño debe tenerse en cuenta y aprovechar convenientemente la herencia, definiendo cada método o cada propiedad lo más arriba posible en la jerarquía de clases, para minimizar la redundancia. Además, el diseño debe ser muy general, para asegurar la reutilización futura de las clases diseñadas, haciendo abstracción en lo posible de las particularidades específicas de la aplicación que se desea construir.

Desde el punto de vista del diseñador de clases, el objetivo fundamental es crear clases cuya interfaz sea clara y comprensible, sin dejar traslucir el menor detalle de la implementación concreta que hay debajo (encapsulamiento), lo que le permitirá en el futuro cambiar las estructuras de datos y los algoritmos sin afectar en lo más mínimo a las aplicaciones que estaban utilizando la versión anterior de la clase. Todo esto, naturalmente, tiene un coste: el esfuerzo necesario para asegurar que las clases sean reutilizables.

Desde el punto de vista del programador que va a utilizar una clase para construir su aplicación, pero que no va a diseñarla, el objetivo fundamental es hacer uso de ella con el mínimo esfuerzo, para lo que espera que la interfaz proporcionada por el diseñador sea comprensible y se adapte a sus necesidades. También esto tiene un coste: la pérdida de eficiencia consiguiente al uso de una herramienta generalizada, en lugar de la programación *ad hoc*.

Durante el diseño de una clase concreta, hay que identificar y especificar sus atributos y sus métodos. Para los primeros, conviene tener en cuenta las siguientes consideraciones:

- El conjunto de los atributos de una clase debe ser
 - Completo (han de contener toda la información pertinente).
 - General (cada atributo debe aplicarse a todos y cada uno de los objetos de la clase).
 - Diferenciado (cada atributo representará un aspecto distinto de la clase).
- Cada atributo tendrá un tipo, que puede ser uno de los siguientes:
 - Atómico predefinido (numérico entero o real, carácter, cadena de caracteres, pixel, etc.)

- Atómico enumerativo (color, día de la semana, etc.)
- Colección (conjunto, lista, etc.)
- Composición (incluye referencias a otros objetos).
- Estructura compleja (formada por combinación de dos o más de los tipos anteriores).

De igual manera se procederá con los métodos, los programas que utilizan o modifican los atributos, cuyo conjunto define el comportamiento de la clase y que se desencadenan por medio de un mensaje. Al diseñarlos, conviene tener en cuenta lo siguiente:

- El conjunto de los métodos de una clase debe ser:
 - Completo (deben realizar toda la funcionalidad de la clase).
 - General (cada método debe aplicarse a todos y cada uno de los objetos de la clase).
 - Diferenciado (cada método debe ser simple y realizar una sola función).
- Los métodos pueden clasificarse en los siguientes grupos:
 - Modificador (asigna un valor a un atributo de la clase).
 - Selector (devuelve el valor de un atributo de la clase).
 - Funcional (es responsable del comportamiento de la clase, sin modificar ni devolver directamente los valores de los atributos).
 - Aplicable a la clase como tal (crear, destruir) o al objeto.

6.1. Metodologías de análisis y diseño

Existen diversas metodologías de análisis y diseño orientados a objetos. Prácticamente, cada autor que ha abordado el tema ha propuesto la suya. Desgraciadamente, aún no existe una que presente ventajas suficientes sobre las demás para convertirse en la metodología estándar. Las más conocidas y utilizadas aparecen en la figura 4.10.

6.2. Ejemplo de análisis orientado a objetos

Vamos a ver un ejemplo no demasiado complicado de la forma en que puede abordarse el análisis de una aplicación orientada a objetos. Tomaremos el caso de una interfaz de usuario que permita combinar diversas componentes gráficas de los siguientes tipos:

- Ventanas.
- Barras de acción.
- Menús de selección.
- Paneles de diálogo.

La primera cuestión a resolver es la selección de las clases de objetos diferentes con las que vamos a encontrarnos. La lista indicada puede ser la primera aproximación, aunque quizá convendría añadir algunas más. Por ejemplo, un panel de diálogo puede contener campos de diversos tipos:

- Textos fijos.
- Campos de entrada de datos alfabéticos.
- Menús de selección.
- Ventanas de presentación de datos.

Metodología	Autor(es) (ver ref.bibliográf.)
Clase-responsabilidad-colaboración (CRC)	Beck y Cunningham
Diseño orientado a objetos	Grady Booch
Análisis/diseño orientado objetos (OOAD)	Coad y Yourdon
Ingeniería soft.orientado objetos (OOSE)	Ivar Jacobson
Análisis y diseño orientado a objetos	Martin y Odell
Análisis de conducta de objetos (OBA)	ParcPlace
Técnica de modelado de objetos (OMT)	J. Rumbaugh
Análisis sistemas orientado obj. (OOSA)	Shlaer y Mellor
Diseño dirigido por responsabilidad(RDD)	Rebecca Wirfs-Brock
Técnica de modelado visual (VMT)	IBM

Figura 4.10.

Se observará que algunos de los objetos de la segunda lista aparecen ya en la lista anterior. Otros pueden considerarse casos particulares o subconjuntos de aquellos. Por ejemplo, un campo de entrada de datos alfabéticos es un tipo especial de ventana que permite escribir texto.

La segunda cuestión que hay que resolver, una vez tenemos claro con qué objetos vamos a movernos, es la construcción de la jerarquía más apropiada. En definitiva, se trata de ver qué clases de objetos son subclases de otras, y cuáles son las superclases. También es posible que la relación entre dos clases diferentes no sea de inclusión, sino de partición (en el sentido en que el motor de un coche es parte del coche, pero no un subconjunto del mismo). Por ejemplo, algunos de los objetos mencionados podrían clasificarse de acuerdo con la figura 4.11, donde las líneas llenas representan relaciones de inclusión y las flechas de trazos indican la existencia de la relación *contiene a*. El tercer paso consiste en tomar las clases de objetos, una por una, y analizarlas independientemente de las demás, tratando de definir sus atributos.

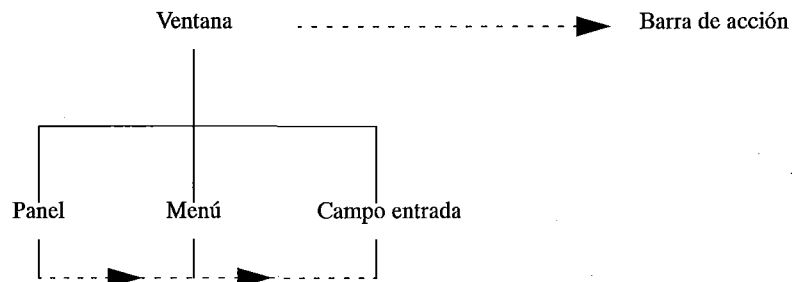


Figura 4.11.

Por ejemplo, en el caso de un campo de entrada de datos alfabéticos, ¿qué atributos necesitaremos? Posiblemente los siguientes:

- Posición del campo, en algún sistema de coordenadas absoluto, como el de la pantalla, o relativo a otra ventana de la que ésta sea una parte.
- Tamaño físico del campo: número de filas y columnas, donde cada fila corresponde a una línea de texto. Alternativamente, este atributo podría definirse en pixels.
- Tamaño lógico del campo, que puede ser mayor o igual que el tamaño físico. En el primer caso, es necesario permitir al usuario *move* por el campo, para hacer visibles las partes inicialmente ocultas.
- Texto del campo. El valor inicial de este atributo es el texto que aparecerá en el campo cuando éste se active. El usuario podría entonces cambiarlo. Cuando un campo se desactive, el valor de este atributo debe ser el texto modificado.
- Color del texto y color del fondo.

- Carácter de relleno: a veces puede interesarnos que el campo de lectura no esté inicialmente lleno de espacios en blanco, sino con otro carácter diferente, cuyo comportamiento, sin embargo, debería ser equivalente al de un espacio en blanco.
- Dirección de la sombra arrojada por el campo, si se desea.
- Definición del funcionamiento de ciertas teclas y del ratón, cuando el campo de entrada de datos está activo.
- Otras particularidades, generalmente representadas por un solo bit de información: si el campo permite al usuario escribir o, por el contrario, es *read-only* (sólo de lectura); en el primer caso, si debe producirse una reorganización automática del texto para no cortar una palabra cuando ésta aparece al final de una línea; si el campo debe dibujarse con un marco alrededor, o no.

Finalmente, y siempre en el contexto del análisis de cada clase por separado, es preciso definir el comportamiento de los objetos que pertenecen a ella. Por ejemplo, en el caso del campo de entrada de datos alfabéticos, podríamos definir los siguientes:

- Un método para dibujar el campo sin darle control.
- Un método para activar el campo, permitiendo al usuario que se mueva por él y, en su caso, modifique la información que presenta.
- Un método para cerrar el campo (borrarlo de la pantalla).
- Métodos para obtener los valores de los atributos del campo.
- Métodos para modificar los valores de los atributos del campo.
- Métodos para obtener información adicional (por ejemplo, la posición del ratón).

Hasta aquí el análisis. El diseño se introduciría un poco más dentro de cada clase, definiendo, por ejemplo, los tipos de datos apropiados para cada uno de los atributos y la forma en que debe ejecutarse cada método. El análisis de algoritmos y la descomposición funcional pueden combinarse aquí con el diseño orientado a objetos.

7. Resumen

En este capítulo se han detallado dos de los principales sistemas que se están utilizando para el desarrollo de aplicaciones de software: la metodología en cascada y la orientación a objetos. En el primer caso se han mencionado algunos de los métodos más utilizados, como el modelo entidad-relación, la descomposición funcional (pusimos el ejemplo de la metodología MAPS) o el diagrama de flujo de datos.

Respecto a la orientación a objetos, hemos definido sus elementos principales (objetos, clases y herencia), sus propiedades esenciales (encapsulamiento, polimorfismo y cierto grado de enlace dinámico), y el hecho de que, en lugar de la llamada de subrutina (instrucción *CALL*), se apoya en el paso de mensajes entre unos objetos y otros.

Se ha visto una breve introducción a la metodología general de análisis y diseño orientados a objetos, y a las muchas formas en que, según los autores, se presenta en la actualidad. Un pequeño ejemplo (cuyo desarrollo completo se sale del alcance del libro) habrá servido para fijar los conceptos más importantes.

8. Notas histórica y bibliográfica

En este capítulo, la descripción de la metodología de diseño MAPS y los conceptos indicados en los apartados 3 y 4 se han tomado esencialmente de Tucker et al. (1992).

La programación orientada a objetos puede considerarse bastante antigua, aunque sólo ha comenzado a imponerse desde finales de los ochenta. Las redes semánticas, introducidas en la psicología cognoscitiva en los años sesenta, y aplicadas en algunos de los primeros sistemas expertos de los setenta (Tema "Lógica", cap. 6, ap. 6) son un precedente. Ya en 1966, el lenguaje SIMULA, definido por el noruego Dahl y sus colaboradores (1966, 1970), contenía prácticamente todos los conceptos posteriormente considerados esenciales en OOP. Sin embargo, SIMULA era un lenguaje de propósito muy restringido (había sido diseñado para aplicaciones de simulación digital discreta) y estos conceptos tardaron en extenderse.

En el decenio de los 70, un equipo de científicos informáticos dirigido por Alan Kay, en Xerox PARC, se apoyó en las ideas de SIMULA para diseñar un lenguaje nuevo, que llamó Smalltalk. Esta vez se trataba de un lenguaje de propósito general, que salió al mercado una década más tarde con el nombre de Smalltalk-80. La empresa Digitalk creó posteriormente su propia versión, Smalltalk-V (ver Digitalk, 1989), que se convirtió en el paradigma de los lenguajes de programación orientada a objetos.

Una vez captada la idea, no tardó mucho en extenderse a otros lenguajes, aparentemente muy alejados de las ideas OOP. Durante los años ochenta, Bjarne Stroustrup (1986) introdujo estos conceptos en el lenguaje C, por entonces el más utilizado en la programación de sistemas, creando una extensión a la que dio el nombre de C++. Se trata esta vez de un lenguaje híbrido, que permite realizar programación clásica, con llamadas de subrutinas (después de todo es una simple extensión de C), pero también es posible limitarse a utilizar sus propiedades orientadas a objetos. Una buena introducción a C++ es Ladd (1990).

Es corriente distinguir entre lenguajes OOP *puros* y lenguajes OOP *híbridos*, que generalmente se construyen como extensión de lenguajes clásicos. Smalltalk se considera puro porque todas las estructuras de datos con las que trabaja son objetos, y sólo se pueden ejecutar programas por medio de mensajes, nunca mediante llamadas a subrutina.

Obviamente, C++ es un lenguaje híbrido o extendido. También lo es Ada 95. En ambos casos, las clases no son otra cosa que tipos abstractos de datos, mientras los objetos son variables pertenecientes a dichos tipos. Muchas de las extensiones introducidas en C para convertirlo en C++ tienen que ver con la conversión de la declaración *struct* en un verdadero sistema de construcción de tipos extensibles. Ada 83 poseía excelentes constructores para la descripción de tipos de datos, por lo que no fue difícil introducir los conceptos OOP en la versión Ada 9X (Skansholm, 1994), recientemente aprobada como Ada 95. Los lenguajes Pascal (Turbo Pascal 5.5) y Modula-2 (Modula-3 y Oberon) se han extendido al mundo de la OOP. También lenguajes tan aparentemente diferentes como el APL pueden extenderse en esta dirección (Alfonseca, 1989).

Es frecuente que muchas aplicaciones comerciales estén escritas en lenguajes extendidos, mientras que los puros desempeñan un papel muy importante para precisar y difundir los conceptos de la programación orientada a objetos. Las diferencias entre unos y otros no se reducen a la terminología (por ejemplo, lo que en Smalltalk se llama *método* en C++ suele denominarse *función miembro* y en Ada 95 es una *operación primitiva*), sino que a menudo afectan a la manera de realizar internamente diversos mecanismos, como la propiedad esencial de la herencia y, sobre todo, el enlace dinámico.

La descripción de la programación orientada a objetos del apartado 5 se apoya esencialmente en Alfonseca y Alcalá (1992). Otros dos libros que dan una excelente introducción son Meyer (1988) y Love (1993). Las técnicas de análisis y diseño orientados a objetos disponen de una amplia bibliografía, como Booch (1994), Coad y Yourdon (1991), Jacobson (1992), Martin y Odell (1992), Rumbaugh y otros (1991), Shlaer y Mellor (1988) o Wirfs-Brock y otros (1990).

Actualmente, la programación orientada a objetos es sólo una parte del conjunto de actividades científicas, técnicas e industriales agrupadas bajo el nombre de *Tecnología de Objetos* (entendiendo tecnología en su acepción anglosajona) u *Objectware*. En este contexto se habla de *objetos distribuidos*, *servidores de objetos*, *depósitos de objetos* y otros conceptos semejantes, lo que ha dado lugar a la necesidad de que objetos de diversos fabricantes puedan entenderse entre sí, para lo que es preciso definir interfaces estándar. Este es el objetivo del *Object Management Group* (OMG), formado por un consorcio de empresas del ramo para definir normas y estándares aplicables a los objetos distribuidos, equiparables a las que patrocinan instituciones públicas y profesionales tan establecidas como el *American National Standard Institute* (ANSI), la *International Standards Organisation* (ISO) y el *Institute of Electrical and Electronics Engineers* (IEEE).

El trabajo del OMG se ha plasmado en la arquitectura CORBA (*Common Object Request Broker Architecture*), al que generalmente se adaptan todos los productos de diversas compañías. Y no es esto todo, pues encima de CORBA, y utilizando como base la tecnología de objetos, han surgido nuevos consorcios y propuestas en campos como la tecnología multimedia, en los que las empresas más potentes ofrecen soluciones alternativas.

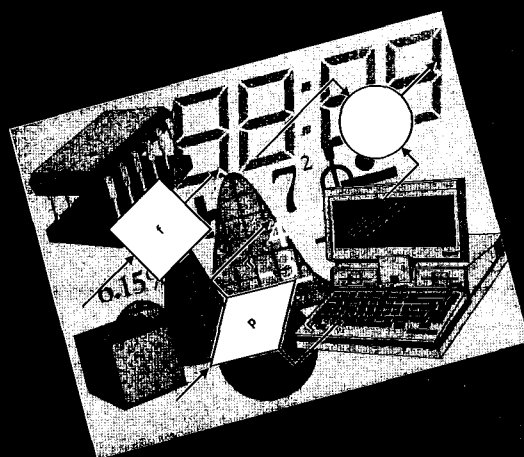
Por último, desde varios puntos de vista, la tecnología de objetos se manifiesta como un movimiento técnico asimilable a un proceso de *evolución cultural* progresivo, que trasciende a la programación para extenderse, como hemos visto,

al análisis, diseño y desarrollo completo de aplicaciones e incluso a la estructura de las bases de datos que éstas utilizan. Esto explica que esta tecnología tienda a integrar o abarcar lo que hemos llamado *programación tradicional* (métodos = programas, algoritmos), y ya ha comenzado a incorporar asimismo otros estilos y lenguajes de programación, como demuestra el caso de la extensión orientada a objetos de Lisp (CLOS), e incluso se habla de sistemas operativos orientados a objetos.

...

Fundamentos de informática

5



Máquina de Turing: definición, esquema funcional y ejemplos

1. Introducción

El propósito fundamental de este capítulo es definir *qué es, cómo funciona y cómo se diseña una máquina de Turing*. A través de varios ejemplos se familiarizará el lector con la estructura de esta máquina, los alfabetos externo e interno, su programación y las distintas representaciones de la misma (lista de quintuplas, esquema funcional, diagrama de estados) y la representación y manejo de las informaciones en la cinta de la máquina.

Pondremos el máximo de énfasis en visualizar configuraciones sucesivas y significativas de la información almacenada en la cinta de la máquina.

2. Definición de máquina de Turing

Una máquina de Turing es un autómata finito, junto con una cinta de longitud infinita (que en cualquier momento contiene sólo un número finito de símbolos) dividida en casillas (cada casilla puede contener un sólo símbolo o estar en blanco) y un aparato para explorar (leer) una casilla, imprimir un nuevo símbolo sobre ella, y mover la cinta una casilla a la derecha o a la izquierda. Veamos en detalle estos elementos:

- a) *La cinta constituye una memoria infinita.* Los símbolos que sobre ella están escritos o se pueden escribir, e_i , $i \in I \subset N$, pertenecen a un conjunto finito E , al que llamamos alfabeto externo de la máquina. Hagamos $m = \text{Card}(E)$. Estos m símbolos sirven para codificar la información suministrada a la máquina. El símbolo blanco, o vacío (\square , generalmente, a no ser que se designe expresamente por 0) forma parte del alfabeto, lo que no querrá decir que pueda considerarse infinita la información contenida en la cinta. Por tanto, en un estadio cualquiera del funcionamiento de la máquina, toda información registrada sobre la cinta se presenta bajo la forma de una palabra escrita en el alfabeto externo E , a razón de un símbolo por casilla.

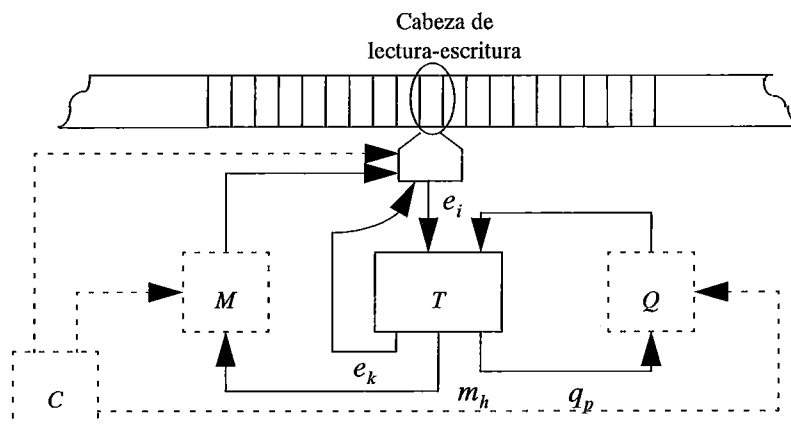


Figura 5.1.

- b) En principio, había una cabeza de lectura y escritura inmóvil y la cinta se desplazaba bajo la misma. Pero en este texto -y no somos los únicos en hacerlo así-, supondremos de ahora en adelante que ocurre lo contrario: *la cinta estará inmóvil y se desplazará la cabeza*, con lo que resultará más sencilla la representación gráfica de la dinámica del contenido de la cinta. Dado que la cabeza se desplaza una posición a la derecha o a la izquierda o no se desplaza, es evidente que las instrucciones que puede ejecutar esta máquina tienen que estar compuestas de las especificaciones de una operación de lectura/escritura y de un movimiento (derecha (\rightarrow), izquierda

(\leftarrow), inmóvil (\leftrightarrow); conjunto M , $M = \{ \rightarrow, \leftarrow, \leftrightarrow \}$. (Observación: puede definirse la M.T. con $M = \{ \rightarrow, \leftarrow \}$, pero aquí se ha optado por la primera versión, más flexible).

- c) El bloque T , que puede atravesar diferentes estados q_i , de un conjunto finito Q , está conectado a la cabeza de lectura/escritura por un enlace de entrada (lectura, símbolo e_i) y un enlace de salida (escritura, símbolo e_k). El estado le es presentado por el bloque Q . La función lógica realizada por el bloque T hace corresponder a la pareja (e_i, q_i) un vector de salida (e_k, m_h, q_p) con $e_i, e_k \in E$, $m_h \in M$, $q_i, q_p \in Q$. Naturalmente, el autómata, en un sentido formal está constituido por los bloques T y Q . Los bloques Q y T actúan como dos memorias internas, que conservan respectivamente el estado q_p y la orden de movimiento m_h , producidos por el autómata durante su trabajo, hasta el comienzo del instante siguiente, comienzo desencadenado por un secuenciador C , que controla los pasos o fases de trabajo. (Nota: aunque se designen por las mismas letras M y Q , no confundir bloque con conjunto. El contexto dirá de qué cosa se trata). En un eje de tiempos se representan las fases del funcionamiento de una M.T. (figura 5.2).

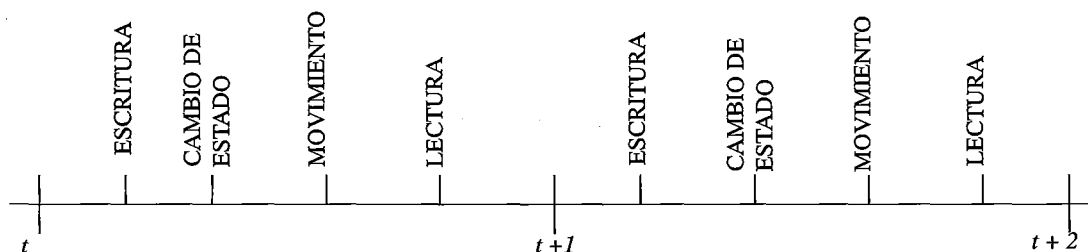


Figura 5.2.

No es esencial el orden en que ocurran "cambio de estado" y "movimiento de la cabeza" (incluso, pueden suponerse simultáneos). Con lo dicho, *podemos definir así el autómata finito $T-Q$* , que es el componente lógico de la máquina de Turing:

$$T-Q = \langle E, (E \times M) \cup (Stop), Q, f, g \rangle \quad (1)$$

$$E \text{ conjunto finito de símbolos en la cinta (alfabeto externo)} \quad (2)$$

$$M \text{ conjunto finito de movimientos de la cabeza} \quad (3)$$

$$Q \text{ conjunto finito de estados internos (alfabeto interno)} \quad (4)$$

$$f: E \times Q \rightarrow Q \text{ función de transición} \quad (5)$$

$$g: E \times Q \rightarrow (E \times M) \cup (Stop) \text{ función de salida} \quad (6)$$

Cualquiera que sea la información, o palabra A escrita inicialmente en la cinta según el alfabeto E , pueden ocurrir dos cosas:

1. Bien, al cabo de un número finito de pasos se detiene la máquina dando la señal de stop. Sobre la cinta se encontrará una cierta palabra B escrita según el mismo alfabeto, que representa la información resultante. Se dice que *la máquina es aplicable a la información A , y que transforma A en B .*
2. Bien, la señal de stop no se produce nunca, en cuyo caso se dice que *la máquina es inaplicable a la información inicial A .*

El funcionamiento de una particular y concreta máquina de Turing podrá describirse especificando los ítems (1) a (6), junto con los siguientes elementos:

Información inicial registrada en la cinta	(7)
Posición inicial de la cabeza	(8)
Estado inicial del autómata	(9)

o, lo que es lo mismo, mediante los ítems (7) a (9) y una tabla (también llamada esquema funcional) de todas las quintuplas posibles, e_p, q_j, e_k, m_h, q_p . Esta tabla es un programa para la máquina de Turing, cuyas instrucciones dicen: "si la máquina está en el estado q_j y lee el símbolo e_i que escriba el símbolo e_k , mueva la cabeza una posición m_h y cambie al estado q_p ".

(Nota: reflexione el lector que en la definición y esquema de la M.T. están presentes de una manera o de otra todos los subsistemas de la figura 2.3 del capítulo 2).

3. Funcionamiento de la máquina de Turing a través de los ejemplos

En este apartado aprenderemos, a través de cinco ejemplos, a analizar máquinas de Turing. En el subapartado 3.3 entraremos en contacto con el procedimiento de composición de máquinas, una especie de diseño modular de máquinas de Turing. A lo largo del apartado nos familiarizaremos adicionalmente con algunos conceptos, tales como la *descripción instantánea* y el cálculo de una M.T.

3.1. Suma de dos números enteros no nulos escritos en el alfabeto $\{ | \}$

La tabla adjunta expresa el programa correspondiente a un algoritmo de suma de los números enteros no nulos para una máquina de Turing, con las siguientes características:

$$E = \{ \square, |, * \}$$

$$Q = \{ q_0, q_1, q_2 \}$$

e_i	q_j	e_k	m_h	q_p
	q_0	\square	\rightarrow	q_2
	q_1		\leftarrow	q_1
	q_2		\rightarrow	q_2
\square	q_0	\square	\rightarrow	q_0
\square	q_1	\square	\rightarrow	q_0
\square	q_2		\leftrightarrow	q_1
*	q_0	\square	\leftrightarrow	Stop
*	q_1	*	\leftarrow	q_1
*	q_2	*	\rightarrow	q_2

$q_j \backslash e_i$	q_0	q_1	q_2
	$\square \rightarrow q_2$	$\leftarrow q_1$	$\rightarrow q_2$
\square	$\square \rightarrow q_0$	$\square \rightarrow q_0$	$\leftrightarrow q_1$
*	$\square \leftrightarrow \text{Stop}$	* $\leftarrow q_1$	* $\rightarrow q_2$

Figura 5.3. Dos representaciones tabulares alternativas

- Información inicial en la cinta: los dos números enteros representados por grupos de | (ejemplo, $3 \equiv |||$; $5 \equiv |||||$) separados por un asterisco*.
- Posición inicial de la cabeza: sobre el primer | de la izquierda.
- Estado inicial del autómata: q_0 .

En el primer instante, la pareja de entrada es $(|, q_0)$, lo que ocasiona un trío $(\square \rightarrow q_2)$, esto es, se borra el primer palote, se desplaza la cabeza una posición a la derecha y se pasa al estado q_2 . Y así una y otra vez. La mejor forma de apreciar la mecánica será con un caso numérico concreto, por ejemplo, la suma de 2 y 3, situado el primer de ellos a la izquierda. Véanse en la figura 5.4 las configuraciones sucesivas de la suma $2 + 3$ en una máquina de Turing.

La máquina se detiene en el movimiento n.º 30. La información resultante, o solución del problema, o palabra B , aparece situada a la derecha de la cabeza cuando ésta se para.

Es evidente que la presentación de las configuraciones de la cinta bajo la forma de la figura 5.4 es poco manejable. Por tal razón, se le prefiere las de la figura 5.5 y 5.6, en las que prácticamente siempre el símbolo "blanco" sería el cero.

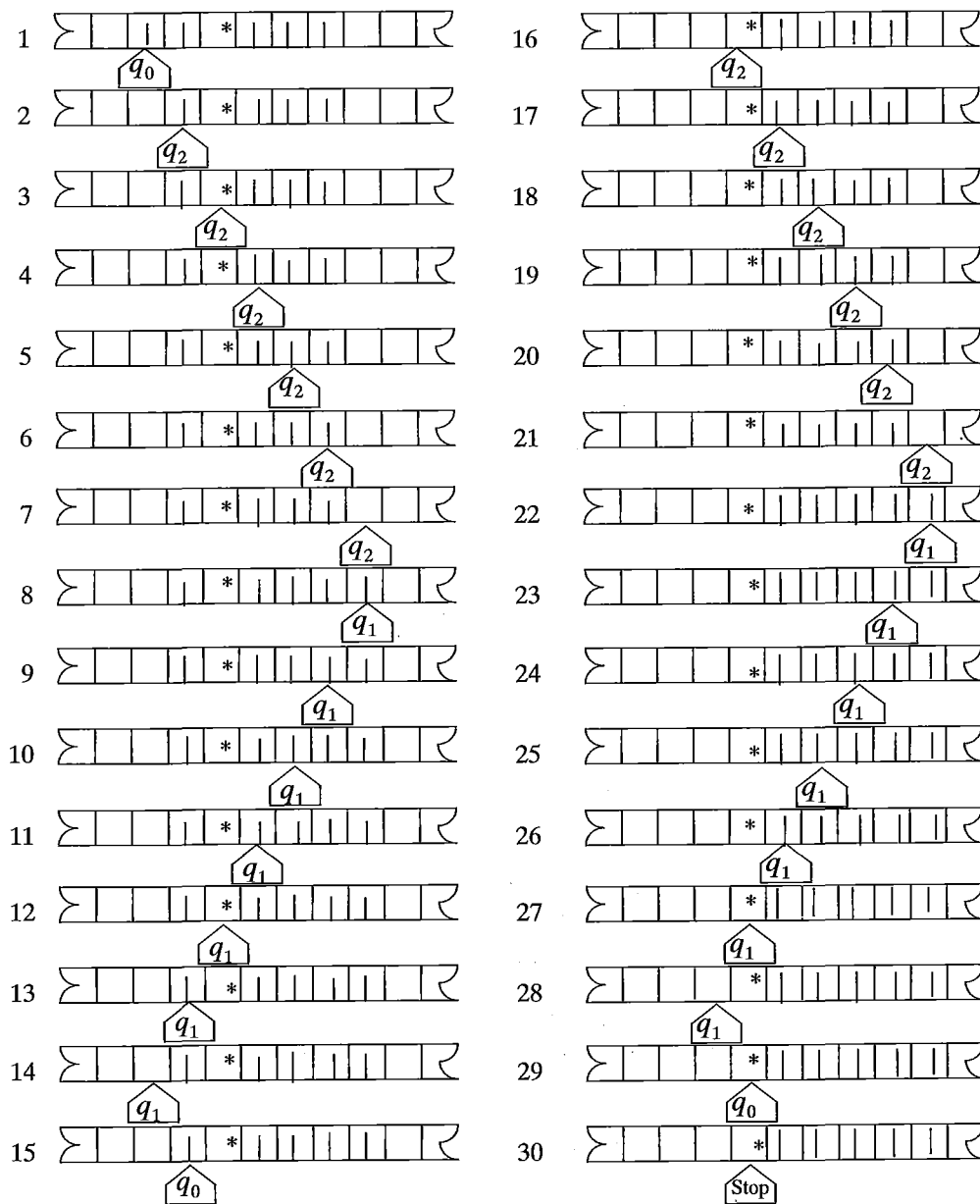


Figura 5.4. Máquina de Turing para sumar dos números enteros no nulos

A las expresiones de la figura 5.6 se les llama *descripciones instantáneas*. La descripción instantánea da de manera absolutamente precisa el comportamiento de la M.T.

En efecto, la descripción realizada, si no se toma en cuenta la variable q_j , nos proporciona el contenido de la cinta y, puesto que el símbolo situado a la derecha de q_j en la expresión es e_i (por este motivo no resulta necesario incluir el subrayado indicativo de la posición de la cabeza), se tiene además la pareja (e_i, q_j) que representa a su vez la situación de trabajo en que se encuentra la máquina.

<u>CINTA</u>	<u>ESTADO</u>
00 * 00	q_0
000 * 00	q_2
000 * 00	q_2
000 * 00	q_2
000 * 00	q_2
000 * 00	q_2
000 * 00	q_2
000 * 0	q_1
000 * 0	q_1
000 * 0	q_1
000 * 0	q_1
000 * 0	q_1
000 * 0	q_1
000 * 0	q_1
000 * 0	q_2
0000 * 0	q_2
0000 * 0	q_2
0000 * 0	q_2
0000 * 0	q_2
0000 * 0	q_2
0000 * 0	q_2
0000 * 0	q_2
0000 *	q_1
0000 *	q_1
0000 *	q_1
0000 *	q_1
0000 *	q_1
0000 *	q_1
0000 *	q_1
0000 *	q_1
0000 *	q_0
00000	stop

Figura 5.5.

Aprovechemos este ejemplo y esta última representación para definir el cálculo de una máquina de Turing. Encontramos esta definición en el cuadro de la figura 5.7.

0	0	q_0		*				0	0
0	0	0	q_2		*			0	0
0	0	0	q_2		*			0	0
0	0	0	*	q_2				0	0
0	0	0	*	q_2				0	0
0	0	0	*	q_2				0	0
0	0	0	*		q_2			0	0
0	0	0	*			q_2		0	0
0	0	0	*				q_1		0
0	0	0	*		q_1				0
0	0	0	*	q_1					0
0	0	0	q_1		*				0
0	0	0	q_1		*				0
0	0	0	q_1		*				0
0	0	0	q_0		*				0
0	0	0	0	q_2		*			0

Figura 5.6.

Cálculo de una máquina de Turing

Se dice que una descripción instantánea γ es *terminal*, relativamente a una máquina Z , si no existe ninguna descripción ω tal que $\gamma \rightarrow \omega$ (\rightarrow significa aquí *es seguida por*).

Se llama cálculo de una M.T. determinada, a una sucesión:

$\gamma_1, \gamma_2, \dots, \gamma_p$
de descripciones instantáneas tales que $\gamma_i \rightarrow \gamma_{i+1}$, para $1 \leq i < p$ con γ_p terminal. Se puede escribir:

$\gamma_p = \text{Res } Z(\gamma_1)$, (10)
lo que se lee: " γ_p es el resultado de γ_1 para la máquina Z ".

Figura 5.7.

Nota: Obsérvese que la expresión (10) es una forma más completa de decir que la máquina Z es aplicable a una determinada información, ya que incluye el estado y la posición de la cabeza en la situación inicial y en la final.

3.2. Algoritmo de Euclides para el cálculo del m.c.d. de dos números enteros escritos en el alfabeto $\{ | \}$

$E = \{ \square, |, *, \alpha, \beta \}$

$Q = \{ q_0, q_1, q_2, q_3, q_4 \}$

- Información inicial en la cinta: los dos números enteros representados por grupos de $|$, separados por un asterisco*.
- Posición inicial de la cabeza: sobre el primer $|$ del número de la izquierda.
- Estado inicial del autómata: q_0 .

El esquema funcional (los recuadros vacíos significan combinaciones imposibles o indiferentes) se representa en la figura 5.8.

$e_i \backslash q_j$	q_0	q_1	q_2	q_3	q_4
$ $	$\beta \rightarrow q_0$	$\alpha \leftrightarrow q_2$	$\beta \leftrightarrow q_1$	$ \rightarrow q_1$	$ \leftarrow q_1$
\square	$\square \rightarrow q_3$	$\square \rightarrow q_4$	$\square \leftarrow q_3$	$\square \rightarrow q_1$	$\square \leftrightarrow stop$
$*$	$\alpha \leftarrow q_0$		$ \rightarrow q_2$	$\square \rightarrow q_2$	
α		$\alpha \leftarrow q_1$	$\alpha \rightarrow q_2$	$ \leftarrow q_3$	$\square \rightarrow q_4$
β	$* \leftarrow q_0$	$\beta \leftarrow q_1$	$\beta \rightarrow q_2$	$\square \leftarrow q_3$	$ \rightarrow q_4$

Figura 5.8. Máquina de Turing para el algoritmo de Euclides aplicado a dos números enteros en $\{ | \}$

La misma información que hay en el esquema funcional puede expresarse mediante un diagrama de estados, que no es, en definitiva, otra cosa que no un organigrama de los que típicamente se usan en programación (figura 5.9) o en la representación de una máquina de Mealy (figura 5.10). (Véanse en cuadro separado los dos tipos de bucles que ocurren en un diagrama de estados).

Sigamos ahora la explicación del proceso de cálculo, para lo que nos puede servir de ayuda la ilustración de algunos instantes significativos en el procesamiento de la cinta en el caso particular de los números 3 (izquierda) y 6 (derecha). (Véase figura 5.11). La primera fase del proceso consiste en suprimir el asterisco que separa los dos números para hacer como si se quisiera yuxtaponer éstos, de manera que formen una palabra escrita en el alfabeto $\{ | \}$, y en disponer la cinta de suerte que quedase bajo la cabeza el último símbolo $|$, que se habrá sustituido previamente por α . Esta fase acaba en el movimiento número 13 (¡atención, esto sólo es cierto en el caso práctico considerado!).

A esta primera fase siguen los ciclos de comparación y sustracción que movilizan respectivamente los estados q_1, q_2 y los q_3, q_4 .

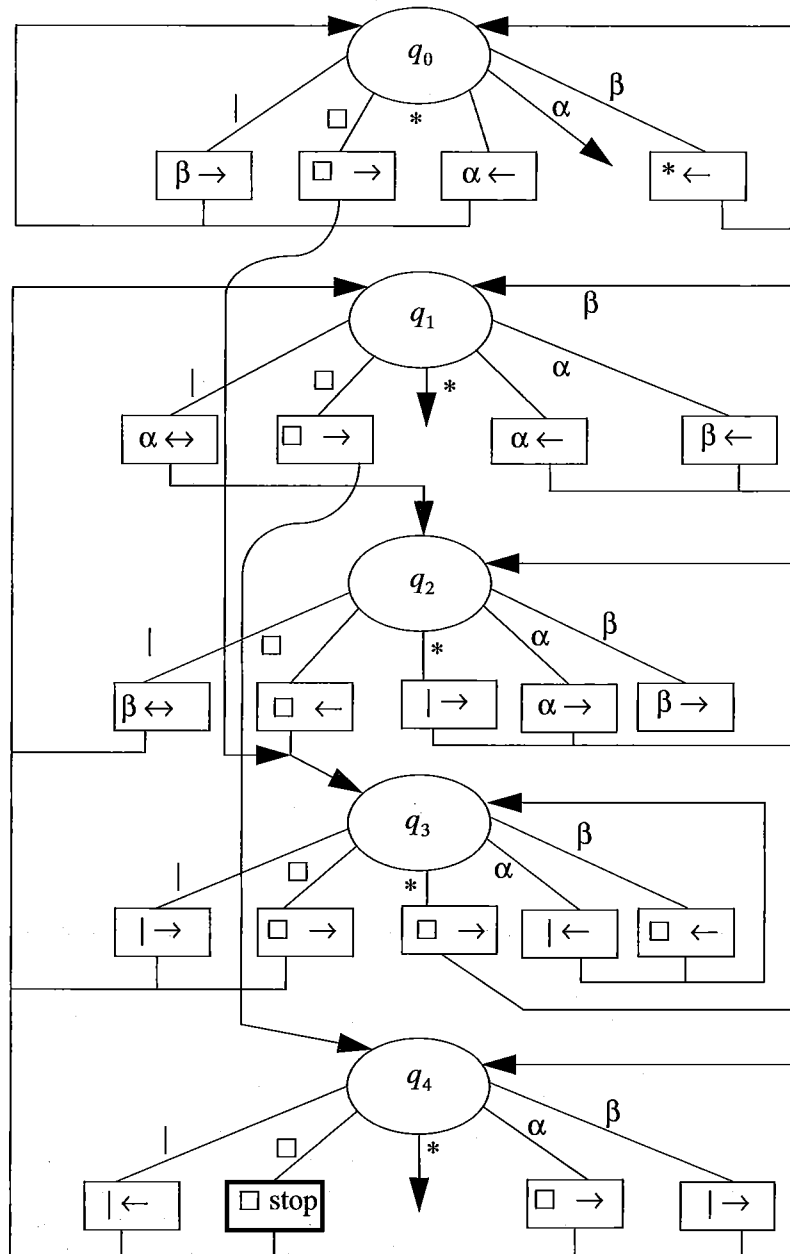


Figura 5.9. Diagrama de estados de una máquina de Turing para el cálculo del m.c.d. de dos números enteros. Los rectángulos representan la acción sobre el contenido y la posición relativa de la cinta

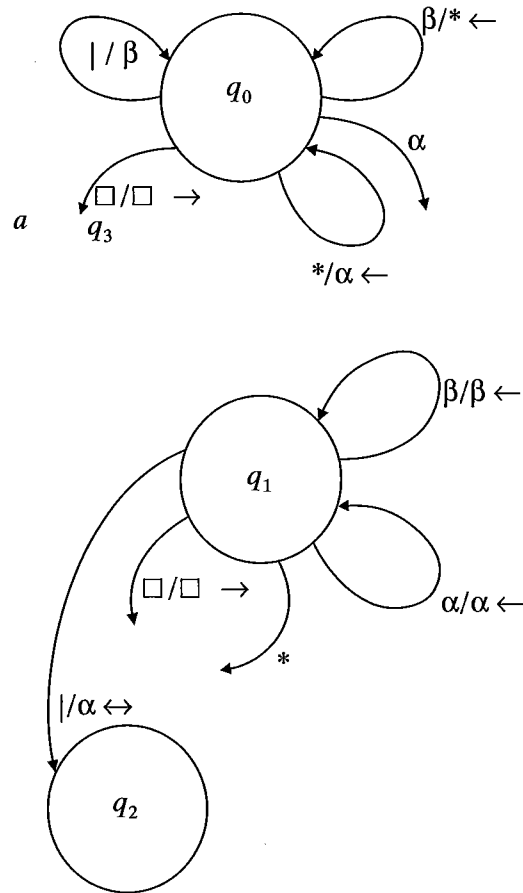


Figura 5.10. Diagrama (incompleto) de estados en forma de máquina de Mealy

Para comparar los dos números y detectar cual de ellos es mayor, la máquina examina los dos grupos de $|$, marcando con un símbolo diferente (α para el número de la izquierda, β para el de la derecha), alternativamente un símbolo $|$ de cada grupo. El punto de partida del ciclo de comparación es el movimiento 13, en que ya se ha sustituido un primer símbolo $|$ del número de la izquierda por α . Dos instantes más tarde (configuración 15), el primer símbolo $|$ del segundo número ha sido sustituido por β . El proceso sigue así, en sustituciones alternadas, hasta la configuración 33, en la que vemos que ya no hay más sustituciones posibles en el primer número. Pero esto la máquina no lo descubre hasta transcurridos 6 instantes más (configuración 39).

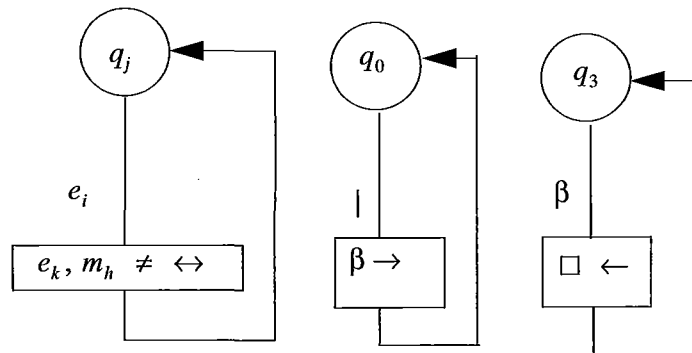
El estado q_4 provoca, cuando en la cinta hay α o β , sucesivos desplazamientos a derecha con sustitución de α por \square (borrado) y de β por 1 , respectivamente. Así se llega a la configuración 46 y un instante después, a la 47.

El diagrama de estados permite poner de manifiesto la existencia de dos tipos de bucles, que son aquellos procesos peculiares en que la máquina puede permanecer repetidamente en un mismo estado:

a)

$e_i q_j e_k m_h q_j$

Ejemplos:

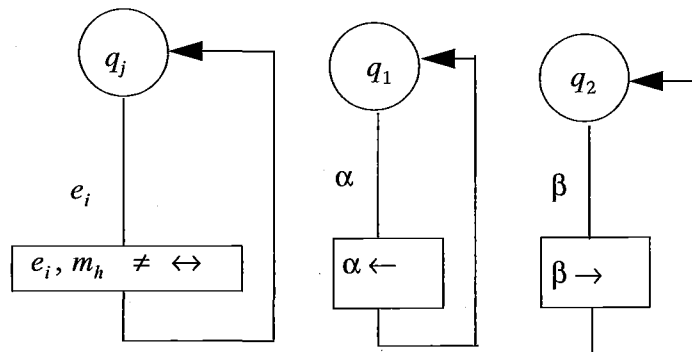


Se sustituye un símbolo e_i por otro símbolo distinto e_k sistemáticamente, mientras queden casillas rellenas con e_i en la dirección de exploración.

b)

$e_i q_j e_i m_h q_j$

Ejemplos:



Se explora la cinta en una determinada dirección sin producir cambios en la información que contiene, mientras la cabeza lea un símbolo determinado e_i .

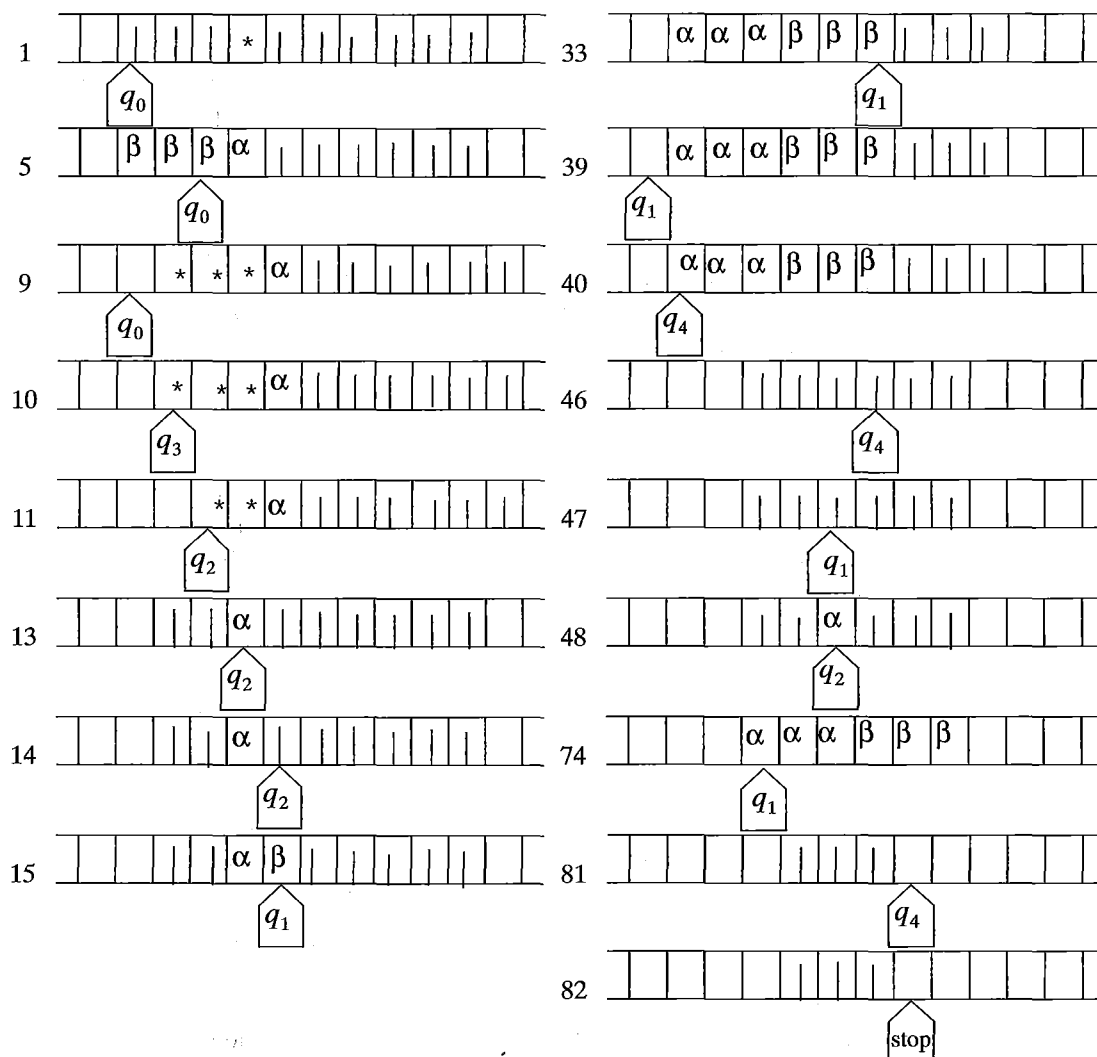


Figura 5.11. Instantes significativos en el proceso de cálculo del m.c.d. de 3 y 6 por una máquina de Turing.

Desde la configuración 40 a la 47 se ha desarrollado un ciclo de sustracción que, si no ha hecho uso del estado q_3 , se debe, una vez más, a las características del caso práctico escogido. (El lector puede ensayar con otro par de números. Así se familiarizará con el funcionamiento de una máquina de Turing y, de paso ¡y con paciencia!, podrá verificar si el programa dado para el cálculo del m.c.d. es o no es correcto).

La configuración 48 marca el principio de un nuevo ciclo de comparación hasta el momento 74. Seguidamente tiene lugar otro ciclo de sustracción, hasta el

instante 81. Las alternancias de ciclos de comparación y de sustracción seguirían desarrollándose hasta llegar a tener dos números iguales, en cuyo momento se acaba el proceso. Esto es precisamente lo que ya ha ocurrido en el caso del 3 y del 6, y en la configuración 82 se detiene la máquina, quedando el resultado escrito a la izquierda de la cabeza: 3.

3.3. Cálculo del m.c.d. de dos números enteros escritos en D ($D = \{0, 1, 2, 3, \dots, 9\}$) por el procedimiento general de construir un programa a base de subprogramas. Composición de máquinas de Turing

Ahora nos podemos proponer definir una MT., y su correspondiente programa, para calcular el m.c.d. de dos números enteros expresados en $D = \{0, 1, 2, 3, \dots, 8, 9\}$, dando el resultado en el mismo alfabeto.

Esto se puede hacer por el sistema de elaborar un diseño original, considerado el problema en su totalidad, lo que no resulta demasiado fácil. También puede descomponerse el problema en partes más sencillas, resolver éstas y después ensamblar las soluciones en un conjunto coherente. Esta técnica es la de composición de M.T.'s. Vamos a seguir este segundo procedimiento, que es muy común en informática, debido al menos a estas tres razones:

- a) Es más sencillo (pese a todo, podrá comprobarse cómo rápidamente se hace patente un considerable nivel de dificultad).
- b) Es más fiable.
- c) Es más económico, por ser más sencillo, por ser más fiable y porque puede aprovecharse trabajo ya desarrollado anteriormente.

Y aquí, adicionalmente, es más didáctico. *Se utilizará como subprograma básico la M.T. definida en el apartado 3.2.*

La información inicial de la cinta será $N_2 * N_1$, con $N_1, N_2 \in D$. Si se pretende aprovechar el diseño de M.T. de 3.2 será necesario crear otros subprogramas (de manera más rigurosa, otras M.T.) con la secuencia general expresada por el organigrama de la figura 5.12, que representa, en definitiva, un algoritmo ejecutado por cuatro máquinas de Turing ensambladas.

(No se olvide que todas las máquinas de Turing son estructuralmente idénticas, lo que se materializa por el formato constante y el significado de sus instrucciones e_i, q_j, e_k, m_h, q_p . En virtud de ello es posible aspirar a fundir en una sola M.T. varias M.T. distintas, siempre que se compatibilicen sus elementos diferenciales o condiciones de contorno: los alfabetos, las condiciones iniciales, las funciones f, g y las condiciones terminales).

A continuación damos los esquemas de M.T.0, M.T.1 y M.T.3. Para aligerar los contenidos de los esquemas, convendremos en suprimir los símbolos de

escritura y de estado siguiente cuando sean iguales a los de entrada y el de desplazamiento cuando sea \leftrightarrow . (Reproducimos M.T.2, también con este convenio. Una casilla vacía indica una combinación imposible, ya que nunca se puede producir ésta por causa del convenio).

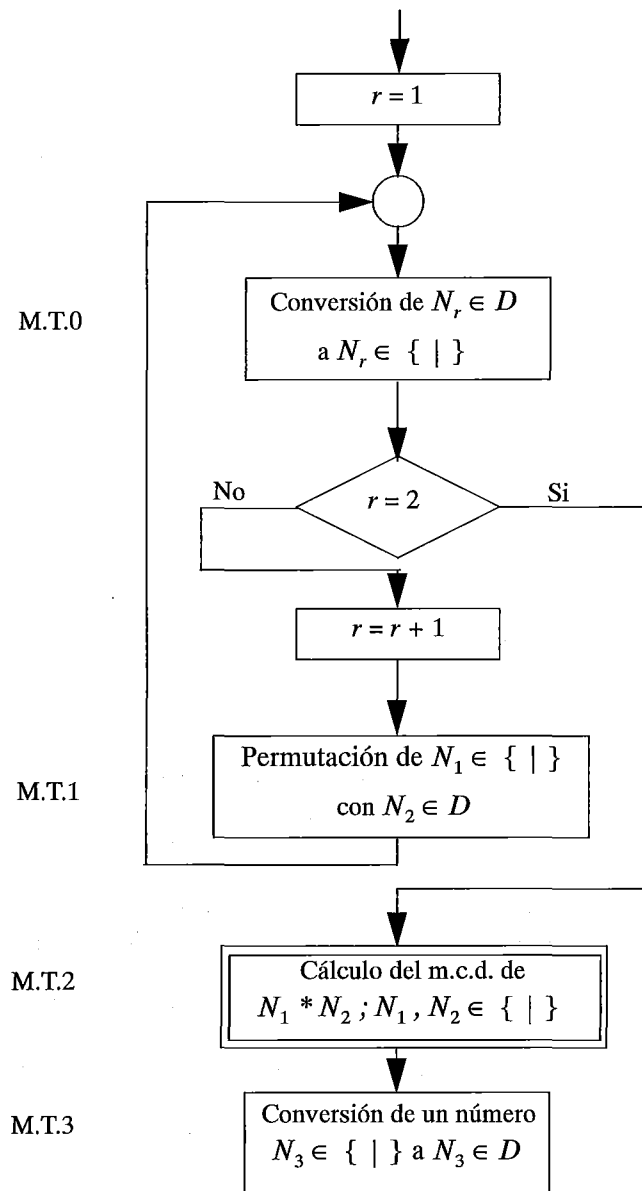


Figura 5.12.

$e_i \backslash q_j$	q_0	q_1	q_2
0	9 \leftarrow	\rightarrow	
1	0 $\rightarrow q_1$	\rightarrow	
2	1 $\rightarrow q_1$	\rightarrow	
3	2 $\rightarrow q_1$	\rightarrow	
4	3 $\rightarrow q_1$	\rightarrow	
5	4 $\rightarrow q_1$	\rightarrow	
6	5 $\rightarrow q_1$	\rightarrow	
7	6 $\rightarrow q_1$	\rightarrow	
8	7 $\rightarrow q_1$	\rightarrow	
9	8 $\rightarrow q_1$	\rightarrow	$\square \rightarrow$
	\leftarrow	\rightarrow	stop
\square	$\rightarrow q_2$	$\leftarrow q_0$	\rightarrow

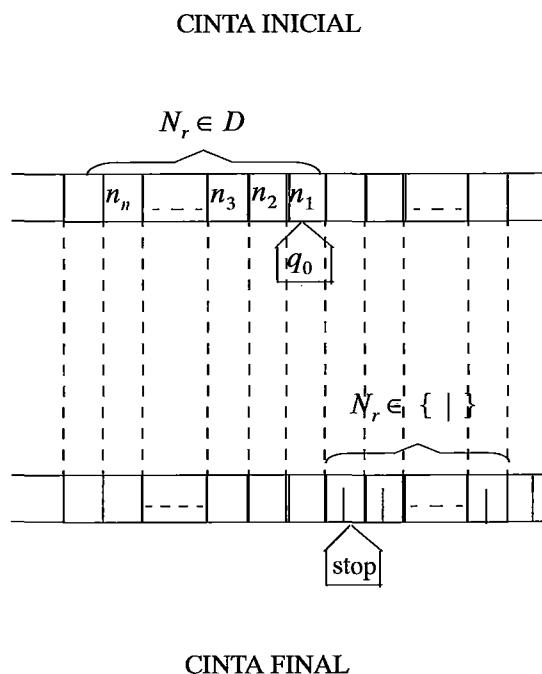


Figura 5.13. M.T.0. Conversión de $N_r \in D$ a $N_r \in \{|\}$.

Se recomienda al lector tome los esquemas M.T.0, M.T.1 y M.T.3, que resuelven tres problemas distintos, como ejercicios independientes de análisis del funcionamiento de máquinas de Turing y, al menos, realice el esfuerzo de captar la idea global del proceso de composición de varias M.T., según se expone a continuación. El esquema de la figura 5.18 puede considerarse como letra pequeña, aunque sería de aconsejar su estudio detallado para aquellos lectores que gusten de profundizar algo más.

Para encajar las cuatro máquinas en una sola y definitiva que resuelva el problema que nos hemos planteado, hay que introducir los cambios oportunos -amén de rebautizar los nombres de los estados al objeto de referirlos a un alfabeto interno más amplio- tendentes a adaptar las diferencias mencionadas entre la forma cómo empieza un subprograma y la forma cómo empieza el siguiente (incluyendo las diferencias de alfabeto externo entre máquina y máquina). La

figura 5.17 recoge la configuración de la cinta de datos antes y después de aplicarle la M.T. correspondiente al proceso indicado en el organigrama de la figura 5.12, con señalamiento expreso de las modificaciones que hay que producir en aquella para ajustar la situación final de una máquina con la inicial de la siguiente.

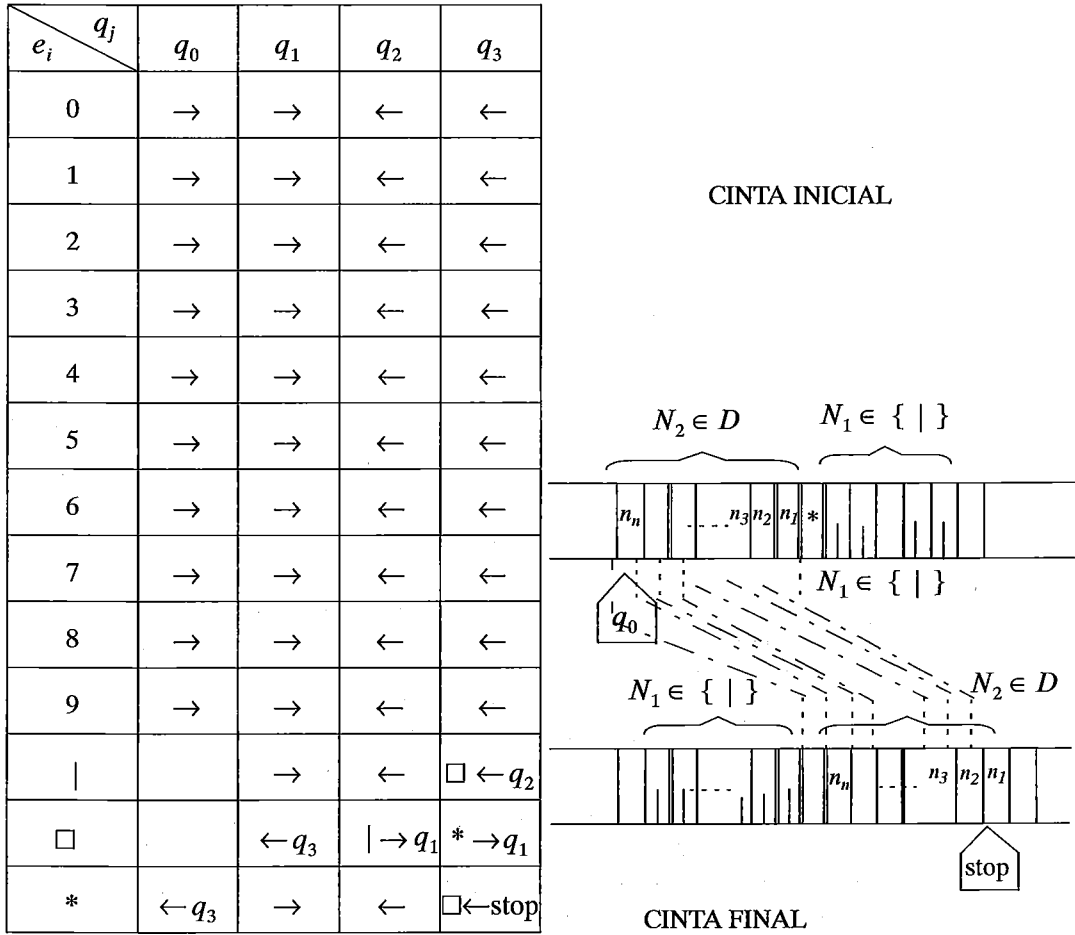


Figura 5.14. M.T.1. Permutación de $N_1 \in \{ | \}$ con $N_2 \in D$.

En la figura 5.18 podemos observar el esquema funcional correspondiente a la máquina de Turing solución (que esperamos sea correcto). Dicha figura utiliza superíndices en los nombres de los estados para que el lector compruebe de qué subprograma procede cada uno de ellos.

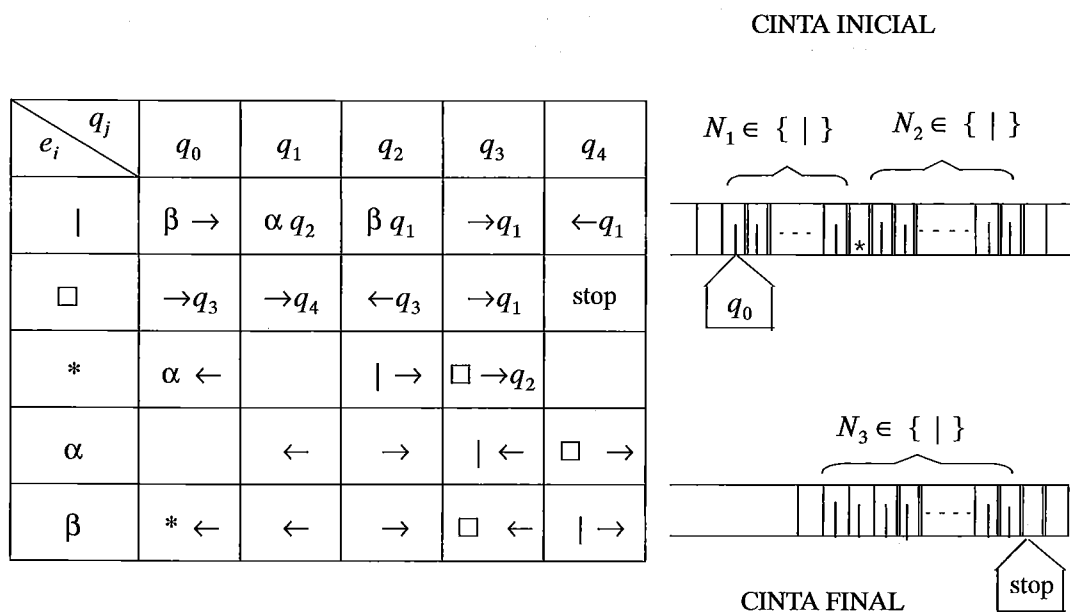


Figura 5.15. M.T.2. Cálculo del m.c.d. de $N_1 * N_2$; $N_1, N_2 \in \{ | \}$

4. Diseño de una máquina de Turing

En el apartado anterior hemos estudiado, partiendo de unos datos en la cinta, el funcionamiento de la máquina ya diseñada. *Otro problema mucho más complejo es el de especificar precisamente las características de una M.T. para resolver un tipo de problema, esto es, definir los conjuntos Q , E , las funciones f , g , y las configuraciones iniciales y finales de la cinta.* Es el problema del diseño, para el cual no existe otra metodología que el propio razonamiento lógico.

En este apartado se verá un nuevo ejemplo de M.T., resaltándose la estrategia seguida en el razonamiento para su creación. Se trata de diseñar una M.T. con cinta direccionable por etiqueta. (Después de estudiar este apartado, se le recomienda al lector reexamine las máquinas de Turing de las páginas anteriores a través de la óptica de diseño).

Suponemos que la cinta es binaria (esto no resta generalidad al problema) y que sobre ella se tienen unos registros de longitud fija, separados por el símbolo \neq . Estos registros están constituidos por una etiqueta, igualmente de longitud fija, y la información, sin que entre ambas exista ninguna separación.

A la izquierda del primer registro, entre el símbolo \neq de éste y otro $=$, se suponen escritos los bits correspondientes a la etiqueta del registro que se quiere localizar, conjunto de bits al que llamaremos "referencia". La cabeza de lectura se sitúa inicialmente sobre el primer \neq , con la máquina en un estado q_0 (figura 5.19).

El trabajo de la máquina consistirá en acudir a la referencia, recordar un bit y sustituirlo por otro símbolo (0 por α , 1 por β). Memorizado este bit, se desplazará la cabeza hacia la derecha en busca del primer registro no explorado y, dentro de él, del bit homólogo. Si coinciden bit de referencia y bit homólogo, éste último deberá ser anulado por la cabeza, que escribirá en su lugar α o β , volviendo a buscar el siguiente bit de referencia.

$q_j \backslash e_i$	q_0	q_1	q_2
0	$1q_2$	stop	\rightarrow
1	$2q_2$	stop	\rightarrow
2	$3q_2$	stop	\rightarrow
3	$4q_2$	stop	\rightarrow
4	$5q_2$	stop	\rightarrow
5	$6q_2$	stop	\rightarrow
6	$7q_2$	stop	\rightarrow
7	$8q_2$	stop	\rightarrow
8	$9q_2$	stop	\rightarrow
9	$0 \leftarrow$	stop	\rightarrow
	\leftarrow	$\square \leftarrow q_0$	\rightarrow
\square	$1q_2$	stop	$\leftarrow q_1$

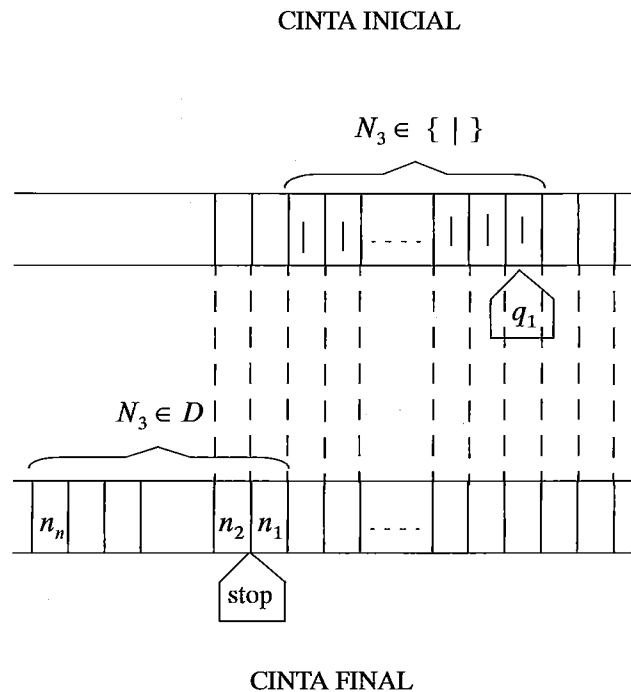


Figura 5.16. M.T.3. Conversión de $N_3 \in \{ | \}$ a $N_3 \in D$

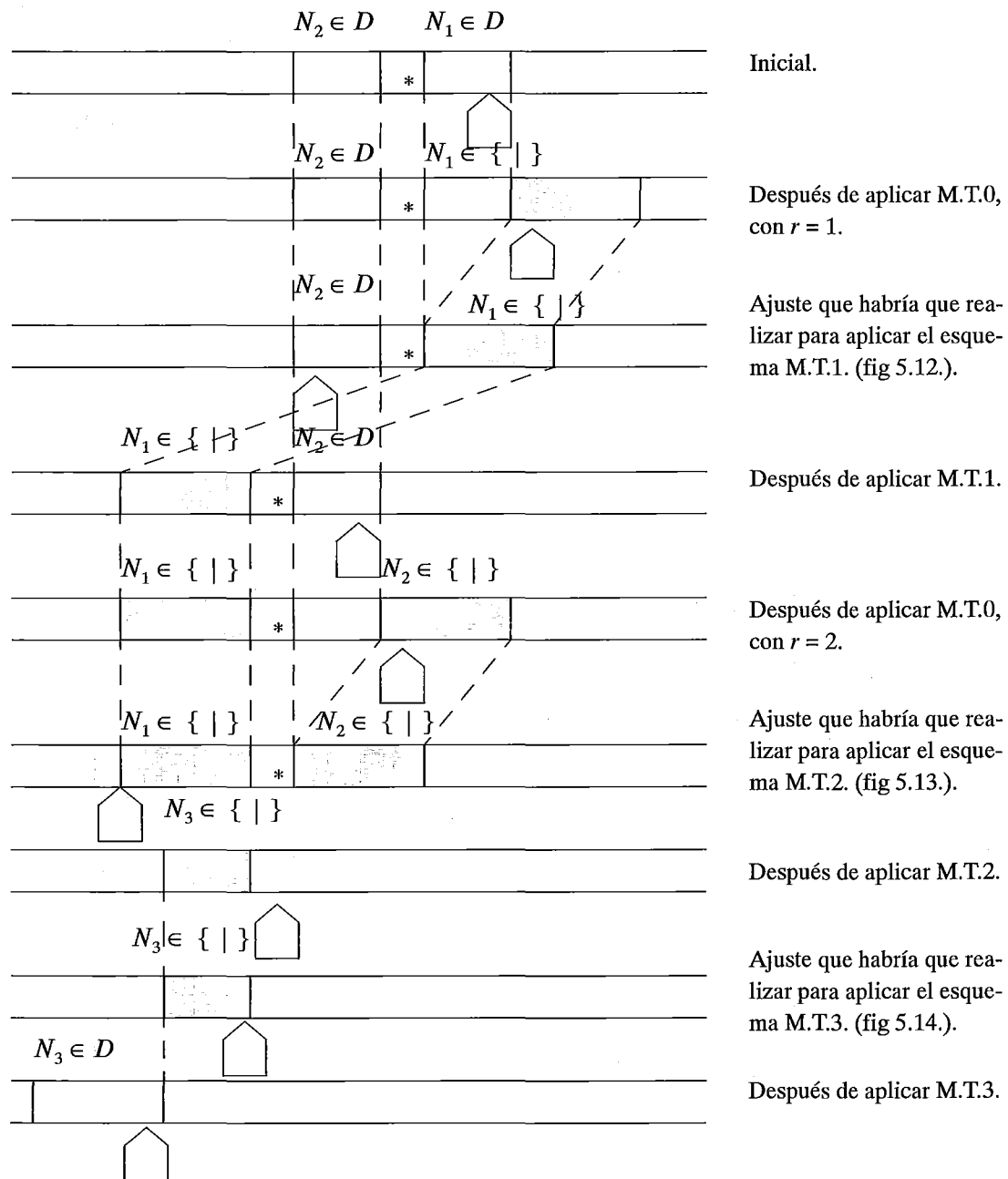


Figura 5.17. Sucesivas configuraciones de la cinta al aplicar el proceso de la figura 5.12

M.T.0'				M.T.1'				M.T.2'					M.T.3'				Nueva denominación de estados
q_j	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}	q_{15}	Denominación parcial
e_i	q_0	q_1	q_2	q_0'	q_1'	q_2'	q_3'	q_0''	q_1''	q_0'''	q_1'''	q_2'''	q_3'''	q_0''''	q_1''''	q_2''''	
0	$9 \leftarrow$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$9 \leftarrow$	\rightarrow					$1 q_2'''$	stop	\rightarrow	
1	$0 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$0 \rightarrow q_1''$	\rightarrow					$2 q_2'''$	stop	\rightarrow	
2	$1 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$1 \rightarrow q_1''$	\rightarrow					$3 q_2'''$	stop	\rightarrow	
3	$2 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$2 \rightarrow q_1''$	\rightarrow					$4 q_2'''$	stop	\rightarrow	
4	$3 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$3 \rightarrow q_1''$	\rightarrow					$5 q_2'''$	stop	\rightarrow	
5	$4 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$4 \rightarrow q_1''$	\rightarrow					$6 q_2'''$	stop	\rightarrow	
6	$5 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$5 \rightarrow q_1''$	\rightarrow					$7 q_2'''$	stop	\rightarrow	
7	$6 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$6 \rightarrow q_1''$	\rightarrow					$8 q_2'''$	stop	\rightarrow	
8	$7 \rightarrow q_1$	\rightarrow		\leftarrow	\rightarrow	\leftarrow	\leftarrow	$7 \rightarrow q_1''$	\rightarrow					$9 q_2'''$	stop	\rightarrow	
9	$8 \rightarrow q_1$	\rightarrow	$\square \rightarrow$	\leftarrow	\rightarrow	\leftarrow	\leftarrow	$8 \rightarrow q_1''$	\rightarrow					$0 \leftarrow$	stop	\rightarrow	
	\leftarrow	\rightarrow	\leftarrow	\leftarrow	\rightarrow	\leftarrow	$\square \leftarrow q_2'$	\leftarrow	\rightarrow	$\alpha q_1'''$	$\beta q_0'''$	$\rightarrow q_0'''$	$\leftarrow q_0'''$	\leftarrow	$\square \leftarrow q_0''''$	\rightarrow	
\square		$\leftarrow q_0$	$\rightarrow q_1'$	q_2'	$\leftarrow q_3'$	$\rightarrow q_1'$	$* \rightarrow q_1'$		$\leftarrow q_0''$	$\rightarrow q_3'''$	$\leftarrow q_2''$	$\rightarrow q_0''$	q_2'''	$1 q_2''''$	stop	$\leftarrow q_1''''$	
*	$\rightarrow q_2$	$\leftarrow q_0$		$\rightarrow q_1''$	\rightarrow	$\leftarrow q_3'$	$\leftarrow q_0$	$\alpha \rightarrow q_2$	$\square \leftarrow q_0''$								
α						$* \rightarrow q_0''$				\leftarrow	\rightarrow	\leftarrow	$\square \rightarrow$				
β										\leftarrow	\rightarrow	$\leftarrow \square$	\rightarrow				

Figura 5.18. Esquema funcional de una M.T. para el cálculo del m.c.d. de N_1 y N_2 ; información inicial $N_2 * N_1$ y $N_2 \in D$

$e_i \backslash q_j$	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
0	\leftarrow	$\alpha \rightarrow q_2$	\rightarrow	\rightarrow	$\alpha \leftarrow q_0$	$\alpha \rightarrow q_6$	$\alpha \rightarrow$	\leftarrow	\rightarrow
1	\leftarrow	$\beta \rightarrow q_3$	\rightarrow	\rightarrow	$\beta \rightarrow q_6$	$\beta \leftarrow q_0$	$\beta \rightarrow$	\leftarrow	\rightarrow
α	\leftarrow	\rightarrow			\rightarrow	\rightarrow		\leftarrow	$0 \rightarrow$
β	\leftarrow	\rightarrow			\rightarrow	\rightarrow		\leftarrow	$1 \rightarrow$
=	$\rightarrow q_1$							$\rightarrow q_8$	
\neq	\leftarrow	stop	$\rightarrow q_4$	$\rightarrow q_5$	\rightarrow	\rightarrow	$\leftarrow q_7$	\leftarrow	$\leftarrow q_0$

Figura 5.20. Máquina de Turing con cinta direccionable

Como es natural, podría haberse seguido la técnica del organigrama para diseñar este proceso de cálculo. A fin de cuentas, el razonamiento lógico-literario que acaba de hacerse, el esquema funcional de la figura 5.20 y el organigrama de la figura 5.22 son equivalentes, si bien la segunda representación es en el caso de la M.T., la más precisa y económica. Los organigramas de las figuras 5.21 y 5.22 son etapas sucesivas de un mismo razonamiento, el último más detallado y con especificación de estados. Para ser coherentes con el capítulo 3, los organigramas son estructurados con arreglo a la técnica de base allí expuesta.

Se le sugiere al lector la resolución del ejercicio siguiente: introducir en el esquema de la figura 5.20 las modificaciones necesarias y mínimas (sin ampliar el número de símbolos) para que la máquina siga realizando la misma función, pero deteniéndose en el símbolo \neq que precede a la primera etiqueta que coincida con la referencia. *Nota:* No deje de observarse que la modificación pedida es del tipo de los ajustes que se han requerido en la composición de máquinas del apartado 3.3.

5. Simulación de máquinas de Turing, máquina de Turing universal y otras consideraciones

5.1. Simulación de máquina de Turing por ordenador

Diseñar una M.T. para resolver un problema medianamente complejo es asunto difícil, pero desarrollar su funcionamiento a base de papel y lápiz es algo absolutamente tedioso.

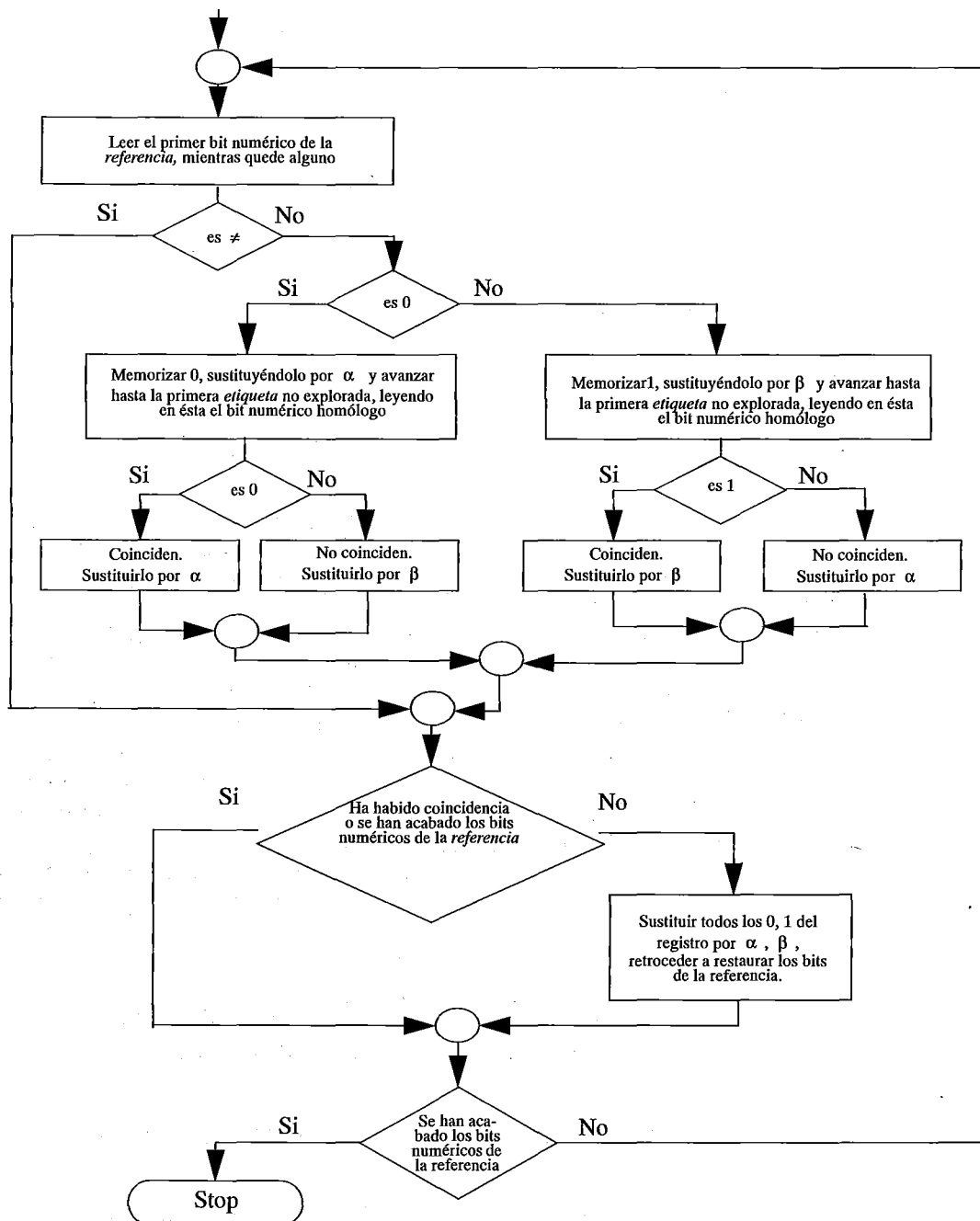


Figura 5.21.

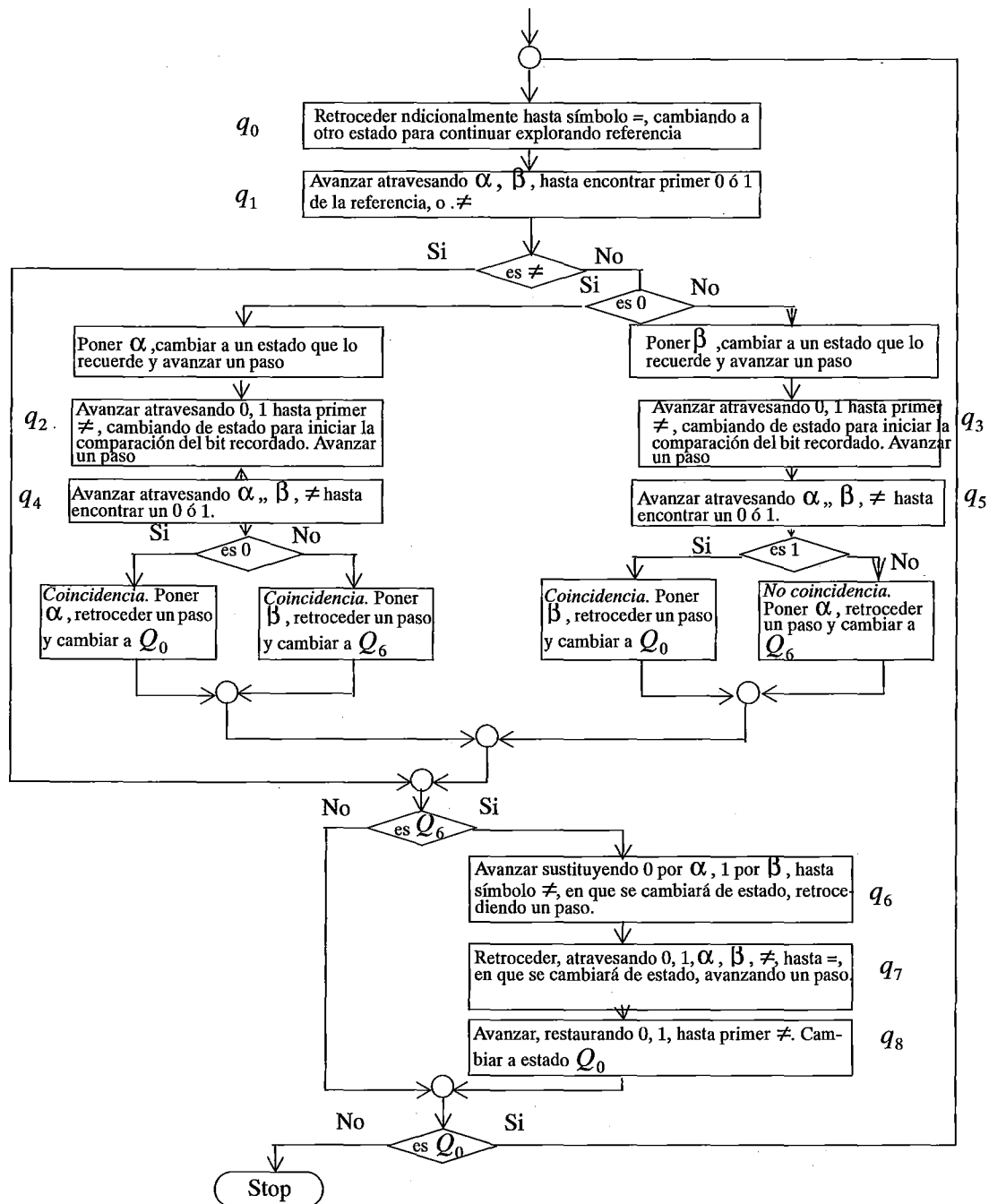


Figura 5.22.

Lo primero podría tomarse como un desafío intelectual, como una prueba (necesaria y no suficiente) demostrativa del grado de capacidad lógica de un futuro especialista en informática, como un juego de sociedad en grupos humanos altamente intelectualizados o como entretenimiento inagotable de náufragos, solitarios y presos que no fueran delincuentes comunes. Lo segundo es un trabajo rutinario con una mecánica que, una vez conocida, no aporta nada especial al individuo e incluso puede resultar psicológicamente casi inaceptable en esta hora en que se dispone fácilmente de ordenadores.

Así pues, simular por ordenador la M.T. es una idea congruente con la eliminación de tarea rutinaria y que permite, dentro de cierto límites, preservar y hasta acentuar los aspectos teóricos de la M.T. Entre otras cosas permite -es un ejemplo- ayudar a poner a punto el diseño de una M.T. como la del apartado 3.3.

Se han escrito algunos programas simuladores de M.T.'s, utilizando un lenguaje de programación con las siguientes características:

DN	Desplazamiento relativo de la cabeza N casillas a la derecha.
IN	Desplazamiento relativo de la cabeza N casillas a la izquierda.
E(e)	Escritura de e en la casilla situada bajo la cabeza, con $e \in E$.
$T(\alpha, e)$	Transferencia condicional. Si el símbolo leído es e , se transfiere control a la instrucción de etiqueta α , si no el programa se desarrolla en secuencia. Cuando no figure símbolo se tratará de una transferencia incondicional.
NOMBRE(A_1, A_2, \dots, A_R)	Nombre de un subprograma donde las A_K son etiquetas de instrucciones o símbolos de la M.T. NOMBRE es la etiqueta de la primera instrucción del subprograma.
END	Corresponde al estado "stop".

Para la representación en el ordenador de los alfabetos de la M.T. se emplearán los propios caracteres admitidos por el ordenador y el único problema es el que se deriva de la limitación física del ordenador respecto de la M.T.: la memoria del ordenador es finita y la memoria de la M.T. es infinita. Por consiguiente, los topes que el programa simulador establezca en la memoria central del ordenador serán los que marquen en cada simulación el espacio de validez de ésta.

Un ejemplo permitirá ver la forma en que habría que describir un esquema de M.T. con el lenguaje especificado más arriba (anotaremos el blanco por una *B*). Tomemos un caso muy sencillo, como el de la M.T. que sumaba dos números enteros no nulos en el alfabeto { | } . (Aquí se sustituirá el palote | por el 1, que es un símbolo admitido por cualquier ordenador). (Véase figura 5.23)..

M0	T(M3, 1) T(M4, B) T(M5, *)
M1	T(M6, 1) T(M7, B) T(M8, *)
M2	T(M9, 1) T(M10, B) T(M11, *)
M3	E(B) D1 T(M20)
M4	D1 T(M0)
M5	E(B) T(M2)
M6	I1 T(M1)
M7	D1 T(M0)
M8	I1 T(M1)
M9	D1 T(M2)
M10	E1 T(M1)
M11	D1 T(M2)
M20	END

Figura5.23.

5.2. Máquina de Turing universal

Todas las M.T.'s pueden ser simuladas por otra máquina de Turing llamada M.T. universal, siempre que se le dé a ésta la información necesaria sobre la primera, a saber:

- Contenido inicial de la cinta.
- Posición inicial de la cabeza.
- Estado inicial.
- Esquema funcional, programa o funciones f y g (como se prefiera llamar).

Los conceptos necesarios para definir una M.T. universal son éstos. Se demuestra que:

1. Cualquier M.T. concreta puede ser simulada por otra con alfabeto binario ($\{0, 1\}$, $\{\square, | \}$) o los dos símbolos preferidos de cada uno. En general, los símbolos de un alfabeto cualquiera son codificables por paquetes de unos, paquetes distinguibles entre sí por paquetes convenidos de ceros).
2. Las posibilidades de una M.T. no se restringen por el hecho de que su cinta sea ilimitada sólo por un extremo.

Una M.T. universal dispone de una cinta ilimitada por ambas partes, con un alfabeto externo $\{0, 1, \alpha, \beta, \neq, \neq, *, b\}$ correspondiéndose α con 0 y β con 1. El símbolo $*$ representa la posición de la cabeza simulada, b es el blanco y al iniciar y terminar la simulación de un movimiento de la M.T. simulada sólo habrá símbolos numéricos $\{0, 1\}$ en la M.T. universal. La información de datos de la cinta simulada se sitúa a partir de una posición a la izquierda de \neq y la información sobre sus estados y funciones a la derecha de ese mismo símbolo. Esta es una versión de una máquina de Turing universal.

Con estos elementos se puede construir una M.T. universal que, combinando las posibilidades de una M.T. de cinta direccionable (apartado 4) y de una M.T. transcriptora de información, puede simular todas las M.T.'s definidas en un alfabeto binario.

5.3. Otras consideraciones

Ya se ha visto que la propiedad de cualquier M.T. de contar con una cinta infinita -característica eminentemente teórica- le confiere una variedad (en el sentido cibernético de esta palabra) adaptable potencialmente al control de cualquier circunstancia de tratamiento de información.

Es conocido que, cuando se recorre un terreno más práctico como es el uso de los ordenadores, el contar con una memoria principal grande es condición *sine*

qua non para poder tratar, con una determinada velocidad, mayor variedad de problemas y problemas más complejos. Aumentando la capacidad de la memoria principal se incrementa el volumen y complejidad de los problemas que es posible tratar con una máquina concreta y a la inversa. Ahora bien, razones tecnológicas y económicas impiden aumentar todo lo que se desearía la capacidad de las memorias, para un precio y un instante histórico precisos. En todos los casos, la velocidad de la memoria representa un freno a la velocidad de que hace gala el procesador o unidad de cálculo, que tiene la virtud, además, a cada nuevo diseño de ordenador, de poder ejecutar un repertorio más amplio de instrucciones distintas. Lo cierto es que, incluso con una memoria finita, el incremento del número de instrucciones distintas lo que hace es añadir versatilidad y velocidad al tratamiento de los problemas, porque un número extraordinariamente reducido de instrucciones distintas basta para describir cualquier cálculo en este tipo de máquinas.

El tipo de máquinas que definió Turing (antes que se diseñara el primer ordenador, recuérdese) es un ordenador ideal, ya que no depende de ninguna característica física ni le preocupa la velocidad u otra clase de eficiencia.

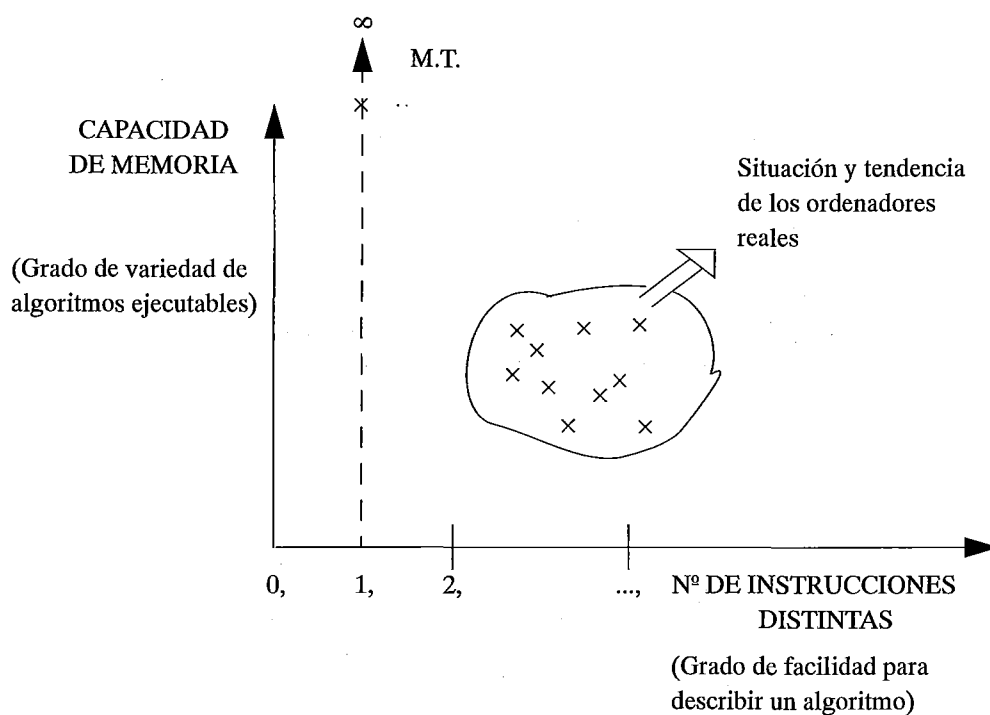


Figura 5.23.

Visto desde la perspectiva de los ordenadores -esto es, a posteriori- es un ordenador con una memoria infinita y una sola instrucción distinta, la quintupla,

tantas veces utilizada en este capítulo. Así pues, no solamente goza de la virtud teórica de poder ejecutar cualquier algoritmo, como, por ejemplo, simular a un ordenador moderno que es una máquina más compleja dotada de un rico repertorio de instrucciones, sino que esto lo hace mediante los pasos más elementales de que se tiene noticia.

La genialidad de Turing consistió en poner su invento fuera de las limitaciones espacio-temporales (espacio para la información, infinito; tiempo, el que sea, pero un número finito de pasos). En tales circunstancias, una sola M.T. es capaz de reproducir el funcionamiento de todas las demás, siempre que disponga de la descripción de las mismas.

Esta última idea, inmanente a la M.T.U., de que una máquina pueda desarrollar procesos más complejos que los que su propia estructura parece permitirle, a condición de que se le suministre la información adecuada, despertó gran interés y ha sido trasladada por analogía al campo de la reproducción biológica para intentar explicar la construcción de la vida y su mantenimiento a partir de las informaciones genéticas.

6. Sucédáneos de la máquina de Turing

Al objeto de que el lector tenga también noticia de ello, conviene que conozca la existencia de ciertos derivados de la versión clásica de M.T., que es con la que hemos venido trabajando en este capítulo. Son, entre otros:

- a) M.T.'s con sólo dos de las tres salidas posibles e_k, m_h, q_p .
- b) M.T.'s con cinta limitada por un extremo.
- c) M.T.'s con más de una cinta y más de una cabeza.
- d) M.T.'s no deterministas.

Se demuestra que ninguna de ellas restringe las posibilidades de la M.T. definida en el apartado 2. Más adelante, mencionaremos la utilidad teórica de algunos de estos tipos de M.T.'s, a propósito de ciertas cuestiones planteadas en el estudio de la complejidad algorítmica (capítulo 7).

7. Resumen

Una máquina de Turing es un artefacto computador constituido por un autómata finito que controla una cinta infinita. Cada paso en el cálculo de una M.T. consiste en escribir un símbolo en la cinta, desplazar la cabeza de lectura/escritura una casilla a la derecha o a la izquierda y asumir un nuevo estado. La acción concreta de cada paso viene determinada por el estado en curso de la máquina y por el símbolo que lee en ese instante la cabeza.

El funcionamiento de una M.T. se especifica completamente por una lista de quintuplas e_i, q_j, e_k, m_h, q_p , donde están todas las combinaciones e_i, q_j que permiten los alfabetos externo E e interno Q , por la cinta con la información inicial y por la situación inicial de la máquina expresada por la posición de la cabeza y el estado del autómata. A la lista de quintuplas se le llama esquema funcional o programa de la M.T.

Distintos ejercicios a lo largo del capítulo han buscado familiarizar al lector con el funcionamiento y las diferentes formas de representar los resultados de una M.T. Una M.T. es capaz de realizar sólo operaciones muy elementales, pero secuencias adecuadas de estas operaciones pueden llegar a componer una amplia variedad de operaciones de manipulación de bloques. Estas últimas operaciones comprenden: formar copias de bloques especificados, sustituir un bloque por otro y comparar bloques. Empleando estas operaciones como subprogramas, es posible diseñar M.T.'s que realizan cálculos muy complejos.

El modelo de M.T. que se ha presentado puede modificarse en varios sentidos sin alterar sus posibilidades últimas como máquina computadora. Tal vez convenga subrayar que, *dado un algoritmo a ejecutar por una máquina de Turing*, en el diseño de ésta -supuesto escogido un modelo específico de M.T. (por ejemplo, con una sola cinta ilimitada por ambos extremos)- *se presenta, en principio, la disyuntiva de disminuir el cardinal del alfabeto externo a costa de aumentar el del interno, o viceversa*. Uno de los resultados más interesantes de esta propiedad es que siempre es posible simular una M.T. por otra M.T. definida sobre un alfabeto externo binario.

Por último, debe resaltarse la *máquina de Turing universal, diseñada para ejecutar un algoritmo de simulación de todas las otras M.T.'s que poseen su misma estructura*. Las especificaciones completas de la M.T. simulada y sus datos de trabajo figuran como datos en la cinta de la M.T.U.

8. Notas histórica y bibliográfica

La máquina de Turing se conoció, como ya se dijo en el primer capítulo de este tema, a raíz de un trabajo fundamental del matemático inglés A. M. Turing, publicado en 1936. Este trabajo, junto con aportaciones previas de Gödel y otras coetáneas, ha significado un impacto para la ciencia matemática, además de constituirse en argumento básico para lo que la informática puede tener de ciencia.

La confección de este capítulo se ha visto ayudada por algunos artículos y libros publicados casi siempre en los últimos años de la década de los sesenta y toda la década de los setenta. En particular, hay que mencionar el libro de Gross y Lentin (1967), del que se ha tomado la definición de cálculo de una máquina de Turing.

En Corge (1975) hemos encontrado interesante la expresión del proceso de cálculo, iniciada en el subapartado 3.2 con el ejemplo del cálculo del m.c.d., y también el diseño de las máquinas M.T.0, M.T.1 y M.T.3 del apartado 3.3, que hemos adaptado, corregido y simulado.

El ejemplo de diseño de una máquina de Turing con cinta direccionable del apartado 4 se encuentra en Scala, Minguet (1974). Asimismo, la versión de máquina de Turing universal del subapartado 5.2. Un desarrollo de la M.T.U. un poco diferente, aunque simple y bastante detallado, se hallará en el capítulo 2 de Hennie (1977).

Los programas simuladores de máquinas de Turing tienen sus épocas, actualizándose a tenor de la tecnología vigente. Por nuestra parte, hemos utilizado la descripción de Curtis (1965). Delgado (1978), cuando terminaba sus estudios, desarrolló bajo nuestra dirección un simulador escrito en Fortran, con el cual pueden simularse máquinas de cierta complejidad. Con él se han simulado las máquinas de los subapartados 3.2 y 3.3.

Disponer de herramientas gráficas de diseño y programación y de entornos operativos de usuario final, del tipo Windows y similares (Macintosh), para ordenadores personales permite hoy día construir simuladores de última generación, muy didácticos, que desplazan la cabeza de la máquina, resaltan en pantalla los nodos y arcos activados en el diagrama de estados, e integran estos esquemas en un documento generado por procesamiento de texto, si así se desea. Citemos, al respecto, el paquete Turing's World, formado por libro y disquete, del Center for the Study of Language and Information (CSLI) (ver Barwise y Etchemendy, 1995) y el simulador, mucho más elemental, de D. Matz (1995). El CSLI produce otros materiales de interés para los lectores de este libro.

A título de curiosidad, tal vez merezca la pena reflexionar acerca de la trascendencia del concepto de máquina de Turing universal acercándose a especulaciones en otros ámbitos disciplinares ajenos a la informática. En tal sentido, cabe mencionar la analogía que habla de construir vida compleja a partir de un núcleo básico y reducido de mecanismos sencillos e informaciones (Singh, 1976, cap.13).

El lector que necesite o desee una presentación más formal y amplia de la máquina de Turing dispone de varias opciones en la bibliografía. Pero si tuviéramos que recomendar aquí un solo texto nos inclinaríamos por el libro de Hopcroft y Ullman (1979), por la simple razón de que, dejando a un lado su incuestionable calidad, tiene la ventaja de que trata conjuntamente la mayor parte de los temas abordados en nuestro libro.

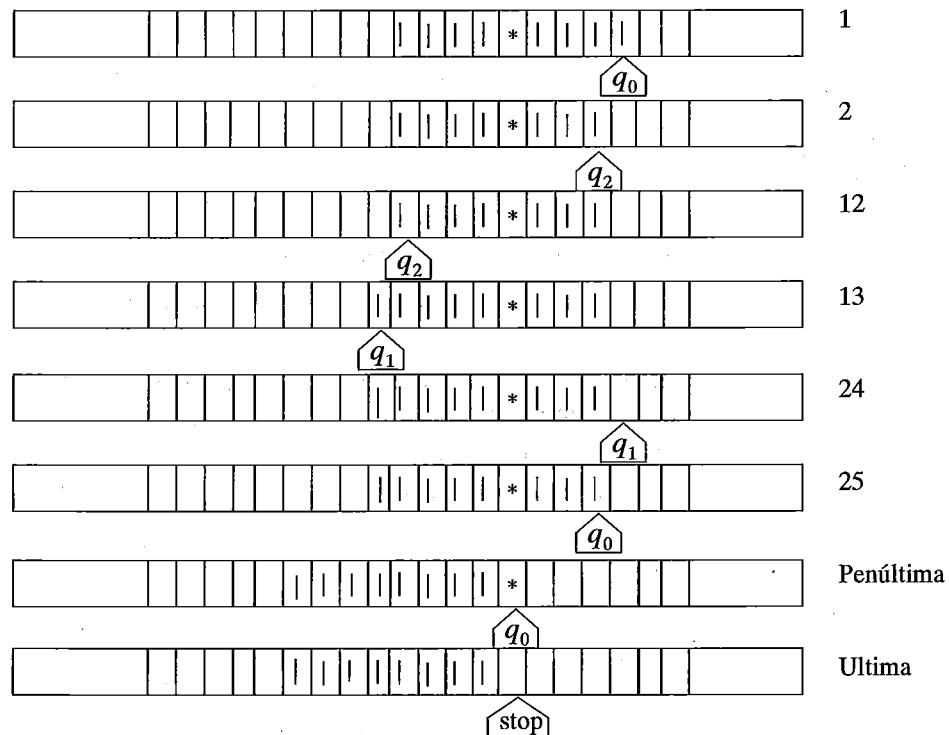
El primero de estos autores ha escrito un artículo de alta divulgación sobre máquinas de Turing en la revista *Scientific American*, posteriormente traducido al castellano (Hopcroft, 1984), que consideramos muy recomendable. Se extiende también en aspectos básicos de computabilidad y complejidad, sobre los que versarán nuestros próximos capítulos.

9. Ejercicios

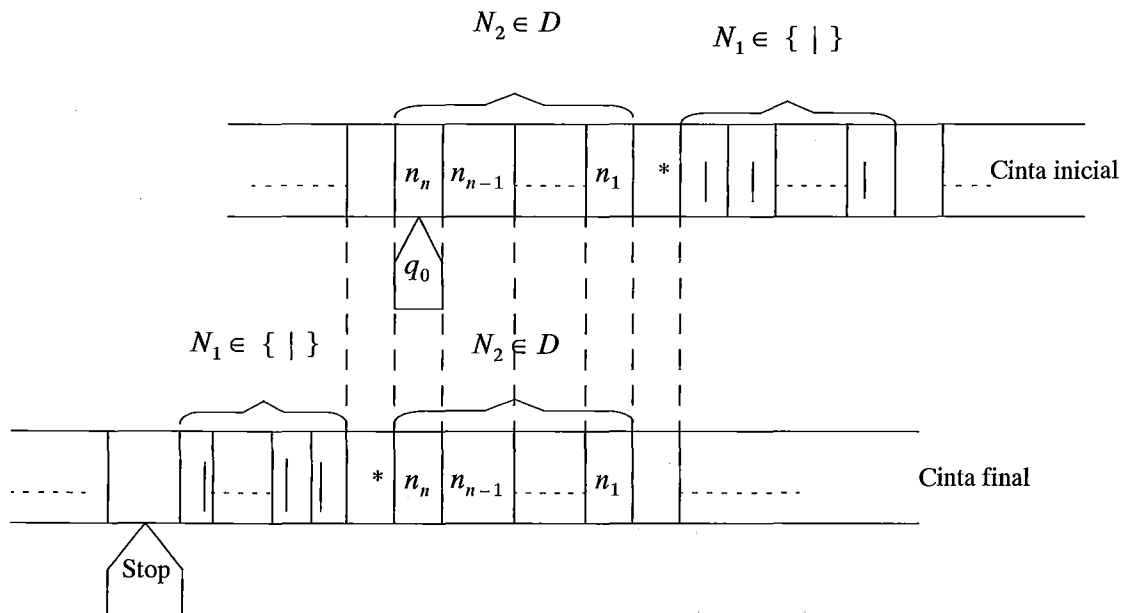
- 9.1. Completar la definición del autómata finito de una máquina de Turing de la que se conoce lo siguiente:
 - El contenido inicial de la cinta es un número entero representado en el alfabeto $\{ | \}$, seguido de un $"*"$.
 - El contenido final de la cinta será el mismo número en binario natural representado en $\{0, 1\}$, seguido de un $"*"$.
 - Conjunto de símbolos de entrada $\{ \square, |, 0, 1, * \}$.
 - Conjunto de movimientos de la cabeza $\{ \rightarrow, \leftarrow, \leftrightarrow \}$.
- 9.2. Escribir, *exclusivamente*, el contenido de la cinta (utilizando el formato de descripción instantánea) de una máquina de Turing para la resolución del algoritmo de Euclides (subapartado 3.2) con dos números enteros en el alfabeto $|$ en los instantes que se indican:

- La máquina se inicia con los números 9 y 3 en la cinta, respectivamente, a izquierda y derecha del asterisco.

	q_0	q_1	q_2
1			
*			



9.4. En la figura 5,14 puede consultarse el esquema funcional de una máquina de Turing que permuta los números N_2 (representado en el alfabeto decimal) y N_1 (expresado en palotes). Partiendo de la misma posición: contenido inicial de la cinta, estado interno y posición de la cabeza, se pide introducir las modificaciones pertinentes en el antecitado esquema funcional para que la máquina realice la siguiente transformación:



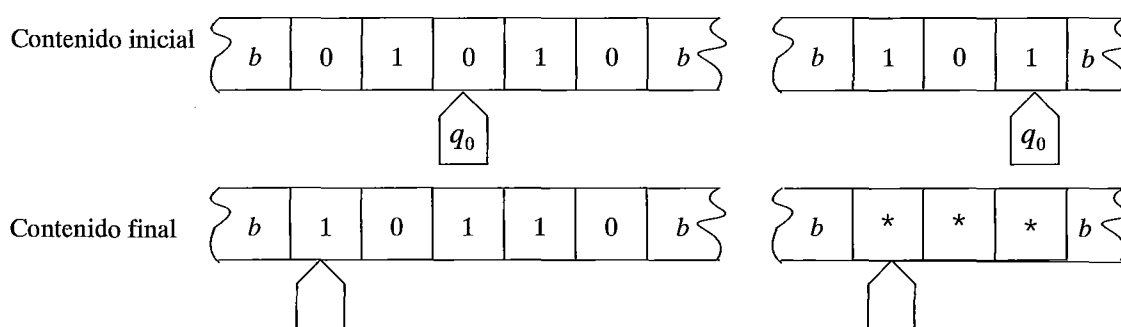
9.5. Diseñar una máquina de Turing cuya especificación es la siguiente:

- Contenido inicial de la cinta: Un número entero positivo representado en complemento a dos y delimitado por casillas en blanco.
- Contenido final de la cinta: La representación en complemento a dos del mismo número negativo.
- Posición inicial de la cabeza: Sobre cualquier símbolo de la representación del número positivo.
- Posición final de la cabeza: Sobre la casilla que contiene el bit de signo.

Notas:

1. El número de casillas que ocupa la representación del número es desconocido. Lo define el contenido inicial de la cinta.
2. En caso de que inicialmente la cinta contenga la representación de un número negativo, la máquina sustituirá los ceros y los unos por asteriscos, para indicar situación errónea.

Ejemplos:



9.6. Se propone introducir las modificaciones necesarias en la máquina de Turing con cinta direccionable, cuyo diseño se explica en el apartado 4, en orden a tomar en cuenta las siguientes condiciones:

- "Ha de preverse un final al cálculo cuando ninguna de las etiquetas de los registros de la cinta coincide con la referencia. Para ello supondremos que en la cinta inicial hay un asterisco (*) inmediatamente a la derecha del último registro".

Se pide introducir en el esquema de la figura 5.2 los cambios pertinentes, añadiendo para ello si es preciso un mínimo número de nuevos estados, de manera que la máquina se detenga sobre el asterisco en el caso en que se produjera la situación descrita más arriba entre comillas. Hágase el esquema funcional completo, recuadrando claramente los cambios o añadidos con respecto al esquema de la figura señalada.

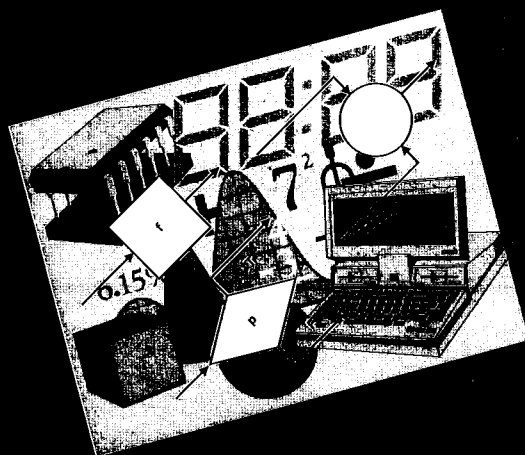
9.7. Diseñar una máquina de Turing que sume enteros no negativos en las siguientes condiciones:

- a) Los números n enteros no negativos, considerados como datos en la cinta, se representarán en el alfabeto $\{ | \}$ con $(n + 1)$ palotes.

- b) Los dos números n_1 y n_2 , que suma la M.T. se presentan en la cinta, de izquierda a derecha, separados por un cero (o blanco \square) y el resto a ceros (o blancos).
- c) La M.T. inicia su operación en el estado q_0 con la cabeza sobre el primer palote a la izquierda.
- d) El resultado de aplicar la M.T. será un bloque de $(n_1 + n_2)$ palotes.
- e) La M.T. termina su operación con la cabeza sobre el primer palote a la izquierda.

Fundamentos de informática

6



6

Máquinas de Turing: algoritmos y computabilidad

1. Introducción

Este capítulo aborda de manera muy esquemática la noción de computabilidad (calculabilidad) en el sentido de Turing (y conceptos relacionados), que formaliza la noción un tanto intuitiva de algoritmo de las definiciones de autores recogidas en el segundo capítulo. Respecto a la definición formal de algoritmo del mismo capítulo ésta es una alternativa más fértil, pues se expresa en términos de una máquina que, no por ser ideal o conceptual, es menos concreta.

2. Función computable y función parcialmente computable

2.1. Hipótesis de Church o de Turing

La idea intuitiva de procedimiento efectivo para desarrollar un cálculo es la misma que la de algoritmo. Pues bien, según la hipótesis de Turing/Church: *la noción intuitiva informal de un procedimiento efectivo sobre secuencias de símbolos es idéntica a la de nuestro concepto preciso de un procedimiento que puede ser ejecutado por una máquina de Turing.*

No existe prueba formal de esta hipótesis pero, hasta la fecha, siempre que en la teoría de las *funciones recursivas*¹ ha sido intuitivamente evidente que existía un algoritmo, ha sido posible diseñar una M.T. para ejecutarlo.

2.2. Función computable

Asociamos una función $F^{(r)}_Z$ con una máquina de Turing A, definiendo $F^{(r)}_Z(n_1, n_2, \dots, n_r)$ como el número $\langle \gamma_p \rangle$ de unos que hay en la cinta cuando Z se detiene, habiéndose iniciado en la siguiente situación.

$$\begin{array}{ccccccc} \dots & 000 & \underbrace{111}_{n_1+1} & \dots & \underbrace{1011}_{n_2+1} & \dots & \underbrace{11011}_{n_3+1} & \dots & \underbrace{10111}_{n_r+1} & \dots & 1000 & \dots \end{array} \quad (1)$$

Si la máquina nunca se detuviera, $F^{(r)}_Z$ no estaría definida para la r-upla considerada.

Intentemos ver más de cerca el significado de los elementos que intervienen en esta función.

En el apartado 5.2 del capítulo anterior se estableció que un alfabeto binario es suficiente -a costa de aumentar el número de estados- para ejecutar cualquier cálculo con una M.T. Con mayor razón podrán tratarse los números enteros no negativos, utilizando un alfabeto $\{0, 1\}$ o $\{\square, |\}$. Escojamos la primera de estas dos representaciones

Se representará un número entero no negativo, n , por $(n + 1)$ unos y a este bloque lo llamaremos \bar{n} , para distinguirlo.

$$\bar{0} = 1, \bar{1} = 11, \bar{2} = 111, \dots, \bar{5} = 11111, \dots$$

¹ Funciones recursivas y funciones computables son equivalentes. El concepto de computabilidad se debe a Turing y el de recursividad a Kleene.

El cero hará las veces de separador, por lo que una r -upla (n_1, n_2, \dots, n_r) se convendrá en representar como en (1), lo que abreviadamente equivale a $\overline{n_1}0\overline{n_2}0\overline{n_3}0\dots0\overline{n_r}$. Si ésta es una información A en la cinta de una máquina de Turing Z , Z será aplicable o no a A . En el primer caso, Z produce el cálculo

$$\gamma_1, \gamma_2, \dots, \gamma_p, \text{ donde } \gamma_1 = q_0\overline{n_1}0\overline{n_2}0\overline{n_3}0\dots0\overline{n_r}$$

y el número entero $\langle \gamma_p \rangle$ es una función que depende de la máquina Z y de la r -upla inicial. Se escribe

$$\langle \gamma_p \rangle = F_Z^{(r)}(n_1, n_2, \dots, n_r) \quad (2)$$

que es una función de valores enteros no negativos, definida sobre N^r o sobre una parte de N^r (función parcialmente definida).

Si se hace ahora al revés, es decir, se parte de una función definida sobre N^r o sobre una parte de N^r se tienen las siguientes definiciones.

2.2.1. Definición de función parcialmente computable

Se dice que una función f definida sobre una parte de N^r es parcialmente computable, para expresar que existe una máquina de Turing Z tal que, para toda r -upla a la que corresponda un valor de f , se tenga:

$$f(n_1, n_2, \dots, n_r) = F_Z^{(r)}(n_1, n_2, \dots, n_r) \quad (3)$$

2.2.2. Definición de función computable

Se dice que una función f es computable para expresar que está definida sobre N^r y es parcialmente computable.

2.3. Ejemplos

2.3.1. La función $f(n_1, n_2) = n_1 + n_2$ definida sobre las parejas de enteros no negativos, que es calculable (computable) en el sentido habitual de la palabra, lo es también en el sentido de la definición 2.2.2. Puede construirse una máquina de Turing Z , tal que

$$n_1 + n_2 = SUM_Z^{(2)}(n_1, n_2) \quad (4)$$

y teniendo en cuenta que $n_1 + n_2 = \overline{n_1} + \overline{n_2} - 2$.

En el momento de detenerse la máquina (4) contiene $(n_1 + n_2)$ unos, cualesquiera que sean n_1 y n_2 . La función f está definida sobre N^2 y es parcialmente calculable, luego es una función calculable.

$e_i \backslash q_j$	q_0	q_1	q_2	q_3	q_4	q_5	q_6
0		$\rightarrow q_5$	q_4	$1 \leftarrow q_4$	$\rightarrow \text{stop}$	$\leftarrow q_6$	
1	$0 \rightarrow q_1$	$0 \rightarrow q_2$	$0 \rightarrow q_3$	\rightarrow	\leftarrow	\rightarrow	$0 \leftarrow q_4$

2.3.2. La función $g(n_1, n_2) = n_1 \div n_2$, sustracción definida sobre el subconjunto $n_1 \geq n_2$ puede demostrarse que es parcialmente calculable sin más que construir la correspondiente M.T., situando, por ejemplo, el número n_1 a la izquierda y el n_2 a la derecha de un cero separador.

La función $n_1 \div n_2$, parcialmente computable podría prolongarse en una función computable $n_1 \div n_2$, definida sobre N^2 , así:

$$n_1 \div n_2 = n_1 - n_2 \quad \text{si} \quad n_1 \geq n_2$$

$$n_1 \div n_2 = 0 \quad \text{si} \quad n_1 < n_2$$

3. Numerabilidad de la colección de todas las M.T.'s

Una M.T. está especificada por una lista de quintuplas e_i, q_j, e_k, m_h, q_p , que forman un conjunto finito. Los valores posibles de i, j, k, h, p son todos numerables. Así pues, la colección de todas las quintuplas es numerable. Consiguientemente, las listas de quintuplas son numerables y, por ende, los autómatas por ellas representados.

Esto significa que las M.T.'s *pueden ordenarse numéricamente*. El problema es cómo escoger un código tal que, dado un número, puedan determinarse las especificaciones de la M.T. correspondiente, si la hubiere, y viceversa.

3.1. Números de Gödel

El método de establecer un código de esta naturaleza, que consiste en *numerizar lo no numérico*, fue propuesto por Gödel antes de que existieran las máquinas de Turing.

Kurt Gödel, lógico eminente desaparecido hace pocos años, escribió en 1930 un artículo que, cuando se publicó en una revista alemana en 1931, produjo el efecto de un paquete de dinamita colocado precisamente en la base de la viga maestra de los fundamentos de la matemática. Fundamentos que, con celo encomiable, estaban renovando los matemáticos de la época, con Hilbert a la cabeza.

Para el lector que no conozca en qué contexto propuso Gödel su técnica de codificación, la preocupación matemática del momento consistía en probar la consistencia de la teoría axiomática de conjuntos, para lo cual Hilbert propuso un programa completo. Pues bien, Gödel probó dos cosas:

1. Si la teoría axiomática de conjuntos es consistente, existen teoremas que no pueden ser probados ni refutados.
2. No existe ningún procedimiento constructivo que pruebe que la teoría axiomática de conjuntos es consistente.

El primer resultado prueba que los problemas no siempre son solubles, ni siquiera en principio; el segundo destruyó el programa de Hilbert para probar la consistencia.

En la teoría axiomática de conjuntos se utilizan símbolos con los que se forman expresiones o cadenas, que son objetos metamatemáticos. Para demostrar sus teoremas, Gödel se sirvió de una codificación numérica de dichas cadenas, que es el asunto que interesa aquí.

Supongamos que se cuenta con los símbolos que a continuación se reseñan. Cada uno de ellos es codificable por los números naturales en la manera indicada

+	-	×	÷	()	=	0	1	2	3	4	5	6	7	8	9	a	b	...	x_i	...
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...	y_i	...

Una de las formas más simples para representar cadenas arbitrarias x_0, x_1, \dots, x_n compuesta con estos símbolos, por ejemplo, sería la de utilizar el número natural

$$p_0^{y_0} p_1^{y_1} \dots p_n^{y_n}$$

siendo p_i el i -ésimo número primo.

La fórmula

se codificaría por el número $2^{12} \cdot 3^1 \cdot 5^{15} \cdot 7^7 \cdot 11^9 \cdot 13^9$.

La fórmula

$$a(a - 1) = aa - a$$

tendría el siguiente número de Gödel: $2^{18} \cdot 3^5 \cdot 5^{18} \cdot 7^2 \cdot 11^9 \cdot 13^6 \cdot 17^7 \cdot 19^{18} \cdot 23^{18} \cdot 29^2 \cdot 31^{18}$.

La cadena $\div - +$ se codifica por $2^4 \cdot 3^2 \cdot 5^1$ igual a 720. Pero, a causa de la unicidad de la factorización en número primos, una cadena puede reconstruirse a partir de su código. Esto es, el número 720, factorizado, nos da $2^4 \cdot 3^2 \cdot 5^1$ que corresponde a la cadena $\div - +$, pues los símbolos que componen ésta tienen los códigos 4, 2 y 1 respectivamente.

En resumen, *cada cadena tiene un número, y números diferentes corresponden a cadenas diferentes*. Disponiendo éstas según el tamaño de sus números de Gödel, puede verse que el conjunto de cadenas es numerable.

3.2. Catálogo de las M.T.'s

Todas las quintuplas que constituyen el esquema funcional de una máquina de Turing forman una cadena de símbolos codificable por un número de Gödel. O, lo que es lo mismo, cada M.T. recibe su número de Gödel z y con él puede catalogarse numéricamente la colección de las máquinas de Turing.

Tratándose de un procedimiento sistemático es posible diseñar M.T.'s con las siguientes propiedades:

- a) *Existe una M.T. que es aplicable a toda secuencia escrita en su cinta (representativa de la lista de quintuplas), transformándola en su número de Gödel.*
- b) *Existe una M.T. que, para todo número escrito en su cinta, proporciona una de estas dos respuestas:*
 - "no existe secuencia correspondiente a este número";
 - "la secuencia de este número es ..." (la lista de quintuplas).

4. De nuevo, la máquina de Turing universal

Se ha visto en el capítulo anterior que una M.T.U. puede simular a todas las otras si se le suministran los datos y las especificaciones de éstas. En el punto 3 acabamos de ver que todas las máquinas pueden ser catalogadas unívocamente, y que puede diseñarse una M.T. tal que, dado un número de Gödel z , entrega como resultado la lista de quintuplas de una M.T., si la hay asociada a ese número. Con esta última perspectiva, es factible pensar en una M.T.U. a la que baste suministrar solamente el número de catálogo de las M.T.'s que se quiera simular. ¿Qué tiene que ver con las funciones computables? Veámoslo.

Llámese X al elemento $(n_1, n_2, \dots, n_r) \in N^r$. A toda función $f(X)$ parcialmente computable puede asociársele el número de Gödel z de la M.T. que

calcula dicha función (es evidente que puede haber varias máquinas que calculen la misma función).

Se define una función $Q_z^{(r)}$:

- Si z es el número de Gödel de una máquina Z que calcula la función parcialmente calculable $f(X)$, entonces $Q_z^{(r)}(X)$ coincide con $f(X)$.
- Si z no es el número de Gödel de ninguna máquina, entonces $Q_z^{(r)}(X)$ toma el valor cero.

Está claro que $Q_z^{(r)}$ es una función cuyo dominio es N^{r+1} y que toma sus valores en N . Puede calcularse por composición de dos máquinas de Turing (véase apartado 3.3 del capítulo 5).

A una M.T. como la definida en el apartado 3.2.b) se le suministra el número z . Si la respuesta es *no*, $Q_z^{(r)}$ toma el valor *cero*. Si la respuesta es *sí*, se alimenta otra M.T. con la secuencia producida por la anterior M.T. que será el programa de cálculo Z , y con el elemento o dato X . El resultado de la ejecución de esta última máquina es $Q_z^{(r)}(X)$, igual a $f(X)$.

Esto es lo mismo que decir que existe una máquina de Turing universal² U , tal que

$$F_Z(X) = F_U(z, X)$$

para todas las M.T.'s Z y todos los enteros z .

4.1. Teorema

Las funciones $Q_z^{(r)}(X)$ son funciones parcialmente computables.

Una M.T. que calcula la función $Q_z^{(r)}(X)$ se llama universal: inscribiendo en su cinta el número z adecuado, ella puede calcular la función parcialmente computable correspondiente a este número.

5. Conjuntos recursivos y recursivamente numerables

En este apartado se va a considerar muy esquemáticamente la aplicación de los conceptos de computabilidad a dos importantes clases de conjuntos de números naturales: los conjuntos recursivos y los conjuntos recursivamente numerables.

²El análogo de U es el ordenador cargado de programas al que se le da el nombre z de uno de éstos y los datos sobre los que tiene que operar.

Hay dos procesos fundamentales de cálculo que pueden asociarse a un conjunto específico G de números naturales. Uno es el proceso de determinar si, dado cualquier número natural x , éste pertenece a G . El otro es el proceso de generar, uno a uno, todas los elementos de G . Ambos procesos están relacionados, pero no son equivalentes.

Para expresar qué es un conjunto de alguna de estas clases debe recordarse previamente la definición de *función característica de G* , C_G , se define así:

$$C_G(X) = \begin{cases} 1 & \text{si } x \in G \\ 0 & \text{si } x \notin G \end{cases}$$

De la misma manera se puede definir la función característica de un conjunto $G \subset \mathbb{N}^*$.

5.1. Conjunto recursivo

Decir que un conjunto es recursivo es expresar que su función característica es computable³.

Esta definición equivale obviamente a decir que existe una máquina de Turing que, aplicada a la información X , la transforma en 1 o en 0. O, en otras palabras, que existe un procedimiento efectivo para decir si un elemento X pertenece o no a ese conjunto.

5.2. Conjunto recursivamente numerable

Un conjunto recursivamente numerable es el dominio de una (por lo menos) función parcialmente computable³.

Dicho en otra forma, significa que existe un procedimiento efectivo para generar sus elementos, uno detrás de otro.

Por ejemplo, el conjunto de los cuadrados de los números enteros es recursivamente numerable -se toman los números 1, 2, 3, 4,... y se van elevando al cuadrado sucesivamente. También es recursivo- dado un número entero cualquiera, se descompone en sus factores primos, viéndose entonces con facilidad si es o no un cuadrado.

³ Podría sustituirse "función computable" por "función recursiva" (véase pie de página del apartado 2.1); y "función parcialmente computable" por "función parcialmente recursiva".

5.3. Dos teoremas más

Enunciamos sin demostración que:

5.3.1. Un conjunto es recursivo si y sólo si el mismo y su complementario son recursivamente numerables.

5.3.2. Existen conjuntos recursivamente numerables que no son recursivos.

6. Determinación de la finitud del proceso de cálculo. Problema de la aplicabilidad

No podemos terminar de hablar de algoritmos y máquinas de Turing sin mencionar un famoso problema que se enuncia así: dada cualquier M.T., una cinta con un número finito de símbolos X y una posición inicial de la cabeza, establecer un algoritmo que permita conocer si el proceso se detendrá o no. "The halting problem" (el problema de la aplicabilidad), como se le conoce en la literatura, es *un problema indecidible*.

Bien, este problema puede enunciarse de una forma distinta. En el enunciado que se acaba de dar se habla de una máquina de Turing, supongamos que su nombre es una vez más Z con el número de orden z . En él se habla también de un algoritmo, así que según la hipótesis de Turing (apartado 2.1), que hemos aceptado, eso es lo mismo que hablar de una M.T. A esta nueva M.T. la llamaremos Z_H .

Veamos un nuevo enunciado, más general, del problema: ¿Existe una máquina de Turing Z_H cuya información en cinta son parejas y que, cuando se le suministra la pareja (z, X) , responde de una de estas formas:

- a) Sí, la máquina Z , de número z , es aplicable al dato X .
- b) No, la máquina Z , de número z , es inaplicable al dato X .

Demostración de que el problema es indecidible.

Supóngase que existe tal máquina de Turing, Z_H .

Sea E un conjunto recursivamente numerable no recursivo (véase teorema 5.3.2). E es el conjunto de definición de una función parcialmente definida calculada por la máquina Z_α .

Sea x un entero cualquiera, ante el que Z_H podría dar una respuesta así:

- "Sí, $x \in E$, Z_α se parará".
- "No, $x \in E$, Z_α no se parará".

Entonces E sería recursivo, contrariamente a la hipótesis. Q.e.d.

Observación.

El problema de la aplicabilidad es trasladable al terreno práctico de la programación de ordenadores. Dados un programa P y los datos D correspondientes, ¿existe un programa general P_H que permita saber si P , aplicado a D , se detendrá? La respuesta es que P_H no existe.

7. Las máquinas de Turing y los lenguajes de tipo 0

La lingüística formal ofrece un campo de interesante aplicación de las M.T.'s. Los distintos lenguajes formales se tratan con cierta extensión en este mismo libro, en el tema "Lenguajes", donde puede verse la clasificación de aquellos en niveles relacionados con las reglas gramaticales que los generan y con los mecanismos que los aceptan (autómatas de uno u otro tipo).

Por lo que se refiere al autómata especial llamado M.T. -asunto que concierne a los capítulos 5 y 6 de este tema- avancemos unas líneas de su relación con la lingüística matemática.

- Puede construirse una M.T. que acepte cualquier lenguaje generado por un sistema de escritura no restringido (lenguajes tipo 0).
- Cualquier lenguaje generado por una gramática del tipo 0 es recursivamente numerable.

Se dice que un lenguaje $L \subseteq E^*$ (esto se verá con detalle en "Lenguajes") es recursivamente numerable si se puede construir una M.T. que acepte todas las cadenas en L y ninguna otra. Un lenguaje es recursivo si tanto L como su complementario, $E^* - L$, son recursivamente numerables. Si L es recursivo se puede construir una M.T. (en la práctica, un programa) capaz de reconocer si una cadena dada es o no un elemento de L . Realmente, esta M.T. estaría compuesta de dos M.T.'s, M.T.1 y M.T.2. El conjunto de ambas o M.T. principal se limitaría a anotar cual de las dos máquinas acepta la cadena (M.T.1 reconocería cadenas en L y M.T.2, cadenas en $E^* - L$).

Supóngase que L es recursivamente numerable, pero no recursivo. En tal caso, no sería posible construir la M.T. principal a que se hacía referencia en el párrafo anterior, al no poderse construir la M.T.2. La consecuencia es que la M.T.1 resultaría insuficiente, ya que, de no pararse, no podría saberse si la máquina (en la práctica, el programa) se había encerrado en un bucle o, simplemente, necesitaba más tiempo para aceptar la cadena en cuestión.

Una línea lingüística como la que acaba de esbozarse se prolonga en importantes desarrollos en el campo de la Inteligencia Artificial.

8. Resumen

Se han definido las *funciones computables* como aquellas a las que corresponde una máquina de Turing aplicable a una r -upla $\overline{n_1}0\overline{n_2}0\overline{n_3}0\dots0\overline{n_r}$, que produce un número $\langle \gamma_p \rangle$ de unos.

Las M.T.'s *pueden catologarse mediante alguna técnica de codificación*, tal que a cada máquina le corresponda un número entero positivo y a cada número entero positivo le corresponda como máximo una sola M.T. La técnica clásica, a nivel teórico, es la de los números z de Gödel.

Así codificada la colección de M.T.'s, pueden diseñarse dos máquinas de Turing, una para codificar máquinas de Turing (es decir, transformar un esquema funcional en número de Gödel) y otra para decodificar (es decir, transformar un número de Gödel en un esquema funcional de M.T., si ésta existe).

Con este instrumental *se ha redefinido la máquina de Turing universal* como aquella a la que sólo es necesario darle el número de catálogo de la M.T. a simular y los datos para ésta.

Las nociones de computabilidad y de parcial computabilidad se emplean en relación con los procesos de cálculo en conjuntos de números naturales para definir qué es un conjunto recursivo (función característica computable) y qué es un *conjunto recursivamente numerable* (dominio de función parcialmente computable).

Por último, se han dado unas muestras del grado de interés de los conceptos de los *conjuntos recursivos y recursivamente numerables* aplicándolos al razonamiento acerca de la *indecibilidad del problema de aplicabilidad de cualquier máquina de Turing y a la aceptación de lenguajes generados por sistemas de escritura no restringidos*.

9. Notas histórica y bibliográfica

El problema de si existe un procedimiento efectivo para determinar si una fórmula arbitraria en el cálculo de predicados de primer orden aplicada a enteros es cierta, parece que estaba ya implícita en Leibniz, se hace explícita en Schröder (1895) y finalmente, en Hilbert, con el cambio de siglo. En 1931, Gödel publicó su famoso *teorema de incompletitud*, probando que tal procedimiento efectivo no podía existir.

La máquina de Turing es el concepto relacionado con la noción intuitiva informal de procedimiento efectivo y se debe a Turing (1936), pero la suposición de que tal noción de función computable pueda ser identificada con la clase de funciones parcialmente recursivas se conoce como la *hipótesis de Church* o la *tesis de Church-Turing* (Hopcroft, Ullman, 1979). Formulaciones alternativas a la de Turing se encuentran en Kleene (1936), Church (1936) o Post (1936).

Formalismos equivalentes a las funciones parcialmente recursivas incluyen el λ -cálculo (Church, 1941), las funciones recursivas (Kleene, 1952) y los sistemas de Post (1943).

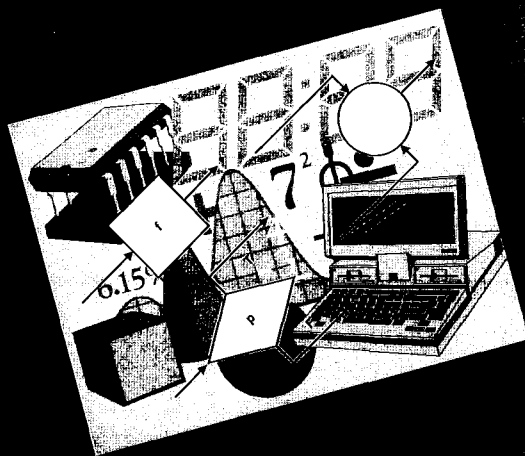
Para este capítulo hemos utilizado con preferencia los libros de Arbib (1965), y de Gross y Lentin (1967). En particular, a este último le debemos la enunciación de los teoremas 5.3.1 y 5.3.2, así como la definición de función característica de un conjunto y el segundo de los enunciados del problema de la aplicabilidad.

Un libro bien escrito sobre introducción a las máquinas de Turing, a las funciones recursivas y temas conexos, es el de Hennie (1977), del que, entre otras cosas, se ha usado su descripción de los procesos fundamentales de cálculo asociables a un conjunto G de números naturales.

La leve referencia del final del apartado 7 a los desarrollos de la línea lingüística dentro de la Inteligencia Artificial se encontró en Hunt (1975).

Fundamentos de informática

7



Complejidad

1. Introducción

Este capítulo aborda el fundamento teórico de las limitaciones prácticas que se presentan en la aplicación del concepto de algoritmo. En pocas palabras, se dedica a estudiar la factibilidad de los algoritmos.

De entre todos los problemas que pueden plantearse, el conjunto de aquellos que son computables (decidibles), es decir, que pueden teóricamente ser resueltos aplicando un algoritmo, es muy reducido. Sin embargo, no todos los problemas computables son factibles en la realidad, por requerir a veces demasiados recursos, ya sean de tiempo, espacio de memoria o circuito materiales.

La teoría de la complejidad algorítmica es la encargada de definir los criterios básicos para saber si un problema computable es *factible* o, dicho de otro modo, si tiene un algoritmo eficiente para su resolución y en este caso cuál es su grado de eficiencia (recuérdese la propiedad de eficacia de todo buen algoritmo, apartado 4 del capítulo 2). Aunque nada más una ínfima parte de los problemas son algorítmicamente factibles, el conjunto es lo suficiente amplio y variado como para que esta teoría resulte no solamente interesante, sino del todo imprescindible para el conocimiento de las nociones esenciales de la informática.

Iremos de lo más general a lo más particular. Primero, se tratará el asunto de la complejidad en términos de la máquina de Turing, puesto que así el tamaño del

problema puede expresarse de la manera más sencilla posible como el número n de posibles casillas que ocupa la información de entrada. Lógicamente, la complejidad del algoritmo resolutor se podrá poner en función de n contabilizando (o estimando por algún procedimiento), bien el número de casillas que es necesario explorar, bien el número de movimientos de la máquina.

Con posterioridad, se trasladará la cuestión a un terreno más práctico, en el que se dispone de operadores más funcionales, por lo general en forma de máquinas computadoras secuenciales, aunque no pueda dejarse sin mencionar la emergencia cada día más patente de máquinas de proceso físicamente concurrentes y su repercusión sobre la complejidad. En todo caso, y al igual que con las máquinas de Turing, *se medirá la complejidad por un orden de magnitud*, del tiempo (por ejemplo), expresado en forma abstracta como *función del tamaño del problema*. Decir "en forma abstracta" significa expresar el tiempo como un número de pasos operativos elementales, al objeto de independizarlo de la velocidad concreta de la máquina ejecutora.

A grandes rasgos, los problemas computables se clasifican en buenos (*complejidad polinómica*) y malos (*complejidad exponencial*). Entraremos en algunos detalles sobre estos conceptos y conceptos derivados.

Para terminar, nos ocuparemos de un apartado de la complejidad teóricamente menor, la complejidad de los programas, a la que se ha dado en llamar -de manera a nuestro juicio incorrecta- *complejidad del software*. La complejidad del software tiene que ver, no con la eficiencia de los algoritmos, sino con la eficiencia de la programación de los algoritmos.

2. Complejidad y máquinas de Turing

Es posible introducir los conceptos involucrados por la complejidad de los algoritmos sin necesidad de apelar a la máquina de Turing, pero su potencialidad de soportar la ejecución de cualquier problema computable la señala como argumento "natural" para aproximarse a criterios universales en lo referente a eficiencia algorítmica.

2.1. Dificultad de encontrar un criterio universal de complejidad

Recordemos que, en efecto, al estudiar la máquina de Turing no se impuso ningún límite de espacio ni tiempo, o, lo que era lo mismo, todo algoritmo que pueda ejecutarse en un número finito de pasos es simulable mediante una máquina de Turing. Esto es así porque la máquina de Turing no padece limitaciones en cuanto a la longitud de su cinta ni en cuanto al tiempo de ejecución.

Como sabemos, en la práctica ocurre que la cantidad de memoria viene limitada por razones diversas, de índole tecnológica, económica u otra. El tiempo de ejecución tampoco puede ser cualquiera, por obvias razones de pragmatismo.

Establecer un criterio universal para poder saber cuán eficiente es un algoritmo no es sencillo a priori, porque un mismo problema puede resolverse sobre diferentes tipos de máquinas, con diferentes grados de eficiencia. Examinemos un momento este asunto por medio de un ejemplo sencillo. Multiplicar dos números de n dígitos decimales por el procedimiento que a todos nos enseñaron en la escuela requiere un tiempo de ejecución proporcional a n^2 . El ejemplo de la figura 7.1.a) muestra un caso en el que $n = 5$. El número de operaciones necesarias es, de una parte, aproximadamente igual a $n \times n$ para obtener las n filas: cada fila resulta de efectuar n multiplicaciones de dos números de 1 dígito decimal. De otra parte, después hay que sumar las n filas interdesplazadas y esto puede costar alrededor de n^2 sumas de dos números de 1 dígito decimal. En resumen, el tiempo total se compone de $c \cdot n^2$ (c , constante) operaciones, que convencionalmente podemos suponer equivalentes a unidades de tiempo, aunque sabemos que hay involucrados dos tipos de operaciones distintas.

(a)

$$\begin{array}{r}
 42013 \\
 \times 27491 \\
 \hline
 42013 \\
 378117 \\
 168052 \\
 294091 \\
 84026 \\
 \hline
 1154979383
 \end{array}$$

A	B	\times	C	D
42	013	×	27	491

$$\begin{aligned}
 A \cdot C &= 1134 \\
 (A + B) \cdot (C + D) - A \cdot C - B \cdot D &= 20973 \\
 B \cdot D &= 6383
 \end{aligned}$$

1154979383

Figura 7.1.

Existen algoritmos más rápidos para resolver el mismo problema. Uno de ellos está aplicado al caso práctico escogido en la figura 7.1.b). Este algoritmo requiere un tiempo proporcional a $n^{1.59}$ y se ha obtenido dividiendo el problema en subproblemas. El algoritmo más rápido conocido a este respecto ejecutable en una

máquina secuencial tiene un tiempo proporcional a $n \cdot \log_2 n \cdot \log_2 \log_2 n$. Si la máquina fuera paralela, el tiempo podría ser proporcional a $\log_2 n$.

En resumen, la eficiencia de un algoritmo no depende sólo del tiempo y del espacio (memoria) utilizados, sino también de los circuitos disponibles (hardware y, en particular, del número y estructura de los procesadores), en suma, de los recursos concretos (véase capítulo 3).

La máquina de Turing nos permite, una vez más, liberarnos de la mayor parte de esas ligaduras materiales gracias a su estructura de operación de formato único (aunque, como sabemos, su estructura admite diversas variantes).

No obstante, utilizaremos la máquina de Turing sólomente para iniciarnos en los conceptos básicos de *complejidad espacial y temporal*, la *complejidad en términos asintóticos* y algunos resultados sobre los efectos en la complejidad debido al *aumento lineal de velocidad*, la *comprensión de la información* y la *reducción del número de cintas*. El estudio detallado de estas cuestiones es objeto de obras especializadas. En el subapartado 2.4 se dará una breve idea sobre la cuestión de los efectos en la complejidad.

2.2. Complejidad espacial. Complejidad temporal

Las definiciones de complejidad algorítmica referidas a máquinas de Turing pueden asociarse con gran generalidad a un tipo de algoritmos tales como los reconocedores de lenguajes. En el apartado 7 del capítulo anterior hemos conocido que existe una relación entre la máquina de Turing y un cierto tipo de lenguajes, a los que se llama "lenguajes tipo 0". El estudio de esta relación y en particular la clasificación de los lenguajes de acuerdo con la complejidad estructural de las respectivas clases de autómatas reconocedores se detalla en el tema "Lenguajes".

Existe otra clasificación de los lenguajes, relativa ésta a la complejidad computacional (o algorítmica), que se basa en la cantidad de tiempo, espacio o cualquier otro recurso necesaria para reconocer un lenguaje en una máquina general, como es la máquina de Turing. Tal complejidad es la que nos interesa aquí.

En el capítulo 6 se usaba una codificación para representar cualquier expresión simbólica por un número, lo que nos llevaba a la posibilidad teórica indudable de catalogar numéricamente la colección de las máquinas de Turing. Por un principio análogo aunque no semejante, podríamos representar los problemas reales, como el de la multiplicación de dos números visto antes, en términos de lenguajes.

Bastaría con definir adecuadamente una función de codificación tal que a cada entrada del problema (en el ejemplo serían los dos números a multiplicar) le correspondiera una determinada cadena de símbolos. Así:

Sea Σ un alfabeto, π un problema y E_π el conjunto de entradas del problema π . Se define la función de codificación α .

$$\alpha: E_\pi \rightarrow \Sigma^* \quad (1)$$

El problema se convierte en un lenguaje L que una máquina de Turing concreta acepta o no. Lícitamente, podemos preguntarnos por la complejidad de las máquinas de Turing (o de los algoritmos) por referencia a los lenguajes.

Se define la complejidad espacial utilizando la máquina de Turing M de la figura 7.2, que consta de una cinta de entrada de sólo lectura con la información enmarcada por delimitadores en sus extremos y de K cintas semi-infinitas de memoria de lectura-escritura. La máquina M tiene una *complejidad espacial* $S(n)$ (o es $S(n)$ -limitada en espacio) si, para cualquier entrada de longitud n , explora como máximo $S(n)$ casillas en cualquiera de las cintas de trabajo.

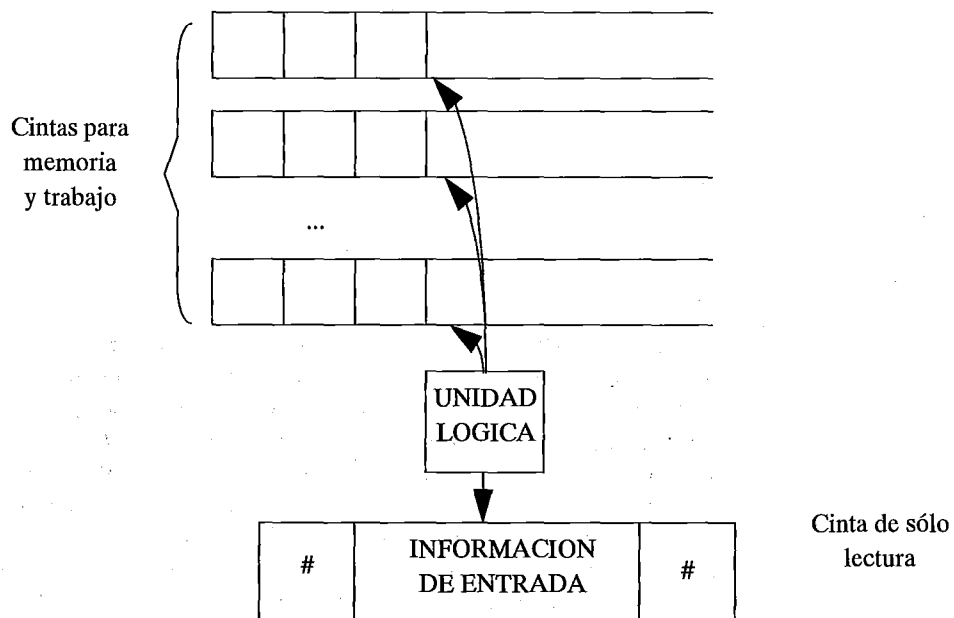


Figura 7.2.

Por razones de conveniencia teórica, la máquina M de la figura 7.3 suministra el modelo adecuado para definir la complejidad temporal de una máquina de Turing. Esta consta de K cintas abiertas por ambos extremos, de lectura y escritura, con una de ellas dispuesta para contener la información de entrada, que, como siempre, suponemos de longitud n . Si para cualquier palabra de entrada de longitud n , M realiza como máximo $T(n)$ movimientos antes de detenerse, se dice que M tiene complejidad temporal $T(n)$ (o es $T(n)$ -limitada en tiempo).

Puesto que la máquina M sirve para ejecutar un algoritmo, lo anterior viene a significar que la ejecución del algoritmo no requiere más de $S(n)$ posiciones en la cinta o más de $T(n)$ desplazamientos, respectivamente.

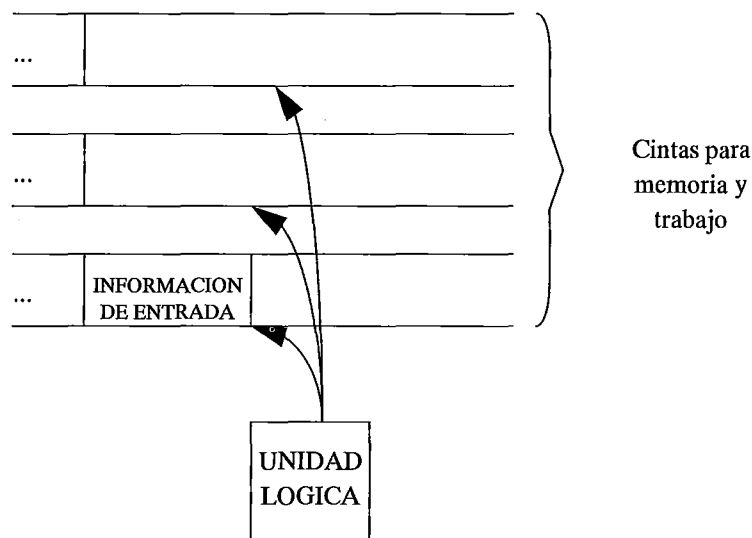


Figura 7.3.

Obsérvese que las funciones $S(n)$ y $T(n)$ no necesitan ser una función exacta, sino sólo un indicador de la forma de variación en función de la longitud de la entrada. Por ejemplo, a efectos de estudiar la eficiencia de un algoritmo simulado por una máquina de Turing M , da igual que su tiempo de ejecución sea $14n^2 + 25n - 4$ ó $3n^2$, pues su forma de crecimiento es en ambos casos de tipo n^2 . Diremos que la máquina M es de complejidad temporal (n^2) , o (n^2) -limitada en tiempo. Esta segunda expresión se comprende intuitivamente, pues si nos fijamos en las funciones anteriores, vemos que en el límite (n infinito) tienden a las asíntotas $14n^2$ y $3n^2$ respectivamente, que son del tipo cn^2 (c , constante).

Así pues, al hablar de complejidad estamos hablando en realidad de *complejidad en términos asintóticos*. Parece razonable asumir que toda máquina de Turing será al menos de complejidad temporal $(n + 1)$, pues éste es el tiempo requerido para leer la entrada. Igualmente, toda máquina de Turing será al menos de complejidad espacial (1) , pues debe poseer al menos una casilla de entrada.

Cuando decimos que una máquina es $T(n)$ -limitada en tiempo, en realidad estamos diciendo $\text{Máx}[(n + 1), T(n)]$ limitada en tiempo, y lo mismo para la complejidad espacial, que será $\text{Máx}[(1), S(n)]$.

2.3. Máquinas deterministas, máquinas no deterministas y tipos de complejidad

Los conceptos que acabamos de ver son aplicables tanto a máquinas deterministas, en las que los pasos a seguir son siempre idénticos para toda entrada, como a máquinas no deterministas, en las que deben tomarse una serie de de-

cisiones que provocan el que la máquina realice más o menos movimientos o escriba más o menos casillas de la cinta.

Para este segundo caso las definiciones son ligeramente diferentes. Diremos que una máquina de Turing no determinista es $S(n)$ -limitada en espacio o de complejidad espacial $S(n)$ si para toda entrada de longitud n , la máquina no debe indagar más de $S(n)$ posiciones de la cinta, cualquiera que sea la secuencia de decisiones que pueda tomar durante la ejecución del algoritmo.

De forma similar, la máquina será $T(n)$ -limitada en tiempo o de complejidad temporal $T(n)$ si, para toda entrada de longitud n , la cabeza de lectura de la máquina efectúa menos de $T(n)$ desplazamientos, cualquiera que sea la secuencia de decisiones.

La familia de lenguajes de complejidad espacial $S(n)$ se denomina DSPACE($S(n)$). Los lenguajes con complejidad espacial no determinista $S(n)$ se engloban en la clase NSPACE($S(n)$). En lo que respecta al tiempo, la familia de lenguajes de complejidad temporal $T(n)$ se denomina DTIME($T(n)$) y si es no determinista NTIME($T(n)$).

2.4. Algunos resultados de la teoría de complejidad en máquinas de Turing

Damos sin demostración determinados resultados. Los primeros vienen a expresar que, en lo referente a la complejidad, es la tasa funcional de crecimiento (es decir, lineal, cuadrática, exponencial) lo que importa, y que pueden desprejiciarse los factores constantes. Podrá hablarse, por ejemplo, de una complejidad $\log n$ sin especificar la base de los logaritmos, ya que $\log_b n$ y $\log_c n$ difieren en un factor constante. En su conjunto, estos resultados vienen a reforzar la idea intuitiva práctica de complejidad asintótica.

Un primer teorema es que si el lenguaje L es aceptado por una máquina de Turing $S(n)$ -limitada en espacio dotada con K cintas de memoria, entonces, para cualquier $c > 0$, L es aceptado por una M.T. $c \cdot S(n)$ -limitada en espacio.

Se prueba simulando la primera M.T. por otra que la simula y que en cada una de sus casillas comprime en un solo símbolo el contenido de r casillas de la cinta correspondiente de la primera M.T.

Otro teorema relacionado con la complejidad espacial en máquinas de Turing dice así: Si un lenguaje L es aceptado por una M.T. $S(n)$ -limitada en espacio provista de K cintas de memoria, es aceptado por una M.T. $S(n)$ -limitada en espacio con una sola cinta de memoria. De hecho, si $S(n) \geq n$ podemos suponer que cualquier M.T. $S(n)$ -limitada en espacio es una M.T. con una sola cinta (que sirve de entrada y de memoria de trabajo).

Por último, puede demostrarse que, bajo ciertas condiciones, si L es aceptado por una M.T. con k cintas $T(n)$ -limitada en tiempo, es aceptado por otra M.T. con k cintas $c \cdot T(n)$ -limitada en tiempo, para cualquier $c > 0$.

Veamos ahora brevemente otro grupo de resultados.

Es indudable que los que en el apartado 6 del capítulo 5 hemos denominado sucedáneos de la máquina de Turing, algunos de los cuales se han utilizado líneas arriba para definir, por conveniencias teóricas, la complejidad, nos proporcionan una idea de la variabilidad de circuitos disponibles (número de cabezas, número de cintas de memoria y sólo lectura, límites en la longitud de las cintas, etc.) y consecuentemente de su influencia sobre la complejidad. Por fortuna, todos estos tipos acaban pudiéndose reducir a una M.T. mono-cinta.

Es así como se demuestra, por ejemplo, que si un lenguaje L está en la clase $\text{DTIME}(T(n))$, es decir, posee complejidad temporal $T(n)$, entonces es aceptado en tiempo $T^2(n)$ por una M.T. mono-cinta.

Asimismo, si L es aceptado por una M.T. $T(n)$ -limitada en tiempo de k cintas, es aceptado por una M.T. de dos cintas de memoria con complejidad temporal $T(n) \cdot \log T(n)$.

Resumiendo, comprobamos que alteraciones importantes de la estructura de una M.T., como puede ser el aumento o disminución del número de cabezas y de cintas de trabajo, si bien no restringen -como se estableció en capítulos anteriores- las posibilidades de la máquina de Turing básica, afectan en cambio a la complejidad temporal. Los dos teoremas que se acaban de enunciar son bien explícitos en este sentido: el paso de complejidad temporal $T(n)$ a complejidades $T^2(n)$ o $T(n) \cdot \log T(n)$ nos ayudan, por analogía, a comprender el cambio cuantitativo que cabe esperar de procesar algoritmos en forma secuencial a procesarlos en forma concurrente (cuando ello es posible).

3. Medidas de la complejidad algorítmica

Desplacemos ahora nuestra atención a terrenos de interés práctico para cualquiera que se dedique a la informática.

En primer lugar, cabe plantearse la pregunta de si realmente medir (o estimar) la complejidad algorítmica tiene algún interés práctico. A diario, numerosos ejemplos nos demuestran que sí. Cualquier software implica cálculos, búsquedas, ordenaciones, manipulaciones u otras operaciones, que son especificados por una solución algorítmica y sometidos a un requisito de eficacia en cuanto a los recursos empleados.

Pueden citarse:

- Programas en tiempo real. La duración de su ejecución está limitada en el tiempo, como es evidente en un sistema que controla el movimiento de una antena que apunta a un satélite para una aplicación de transmisión de datos.
- Programas con volumen limitado de almacenamiento, tal como sucede en un sistema embarcado en un sobrecargado ingenio espacial.
- Programas de uso muy frecuente, por ejemplo, programas de software de base, compiladores, manejadores de dispositivos, y otros. En ellos no tiene

por qué existir estrictamente una limitación de recursos (salvo por razones de competitividad económica), pero, por razones de rendimiento operativo, es en cambio muy importante mejorar los algoritmos utilizados y optimizar, llegado el caso, su codificación.

Como ya sabemos, se ha convenido en que sea el tamaño de los datos de entrada del problema el parámetro para medir su complejidad computacional. Lógicamente, una vez fijado el problema y definida su entrada, la complejidad dependerá de la clase (tiempo, espacio) y número de recursos considerados (máquinas secuenciales, máquinas paralelas) y del algoritmo elegido, y, por último, habrá un aspecto práctico digno de tenerse en cuenta: la variabilidad concreta de la medida de la complejidad para unos datos determinados (una vez fijados los recursos y el algoritmo). En todo caso, la complejidad siempre se expresa en unos términos independientes de la velocidad del operador concreto.

Vamos a empezar viendo los tipos clásicos de complejidad en función del tamaño n de la entrada.

3.1. Complejidad polinómica y complejidad exponencial

La figura 7.4 permite visualizar la variación del orden de magnitud de cuatro algoritmos de complejidad (que supondremos temporal) $\log_2 n$, n , n^2 y 2^n , respectivamente, en función de n .

n , tamaño de datos de entrada	$\log_2 n$ microsegundos	n microsegundos	n^2 microsegundos	2^n microsegundos
10	0,000003 seg	0,00001 seg	0,0001 seg	0,001 seg
100	0,000007 seg	0,0001 seg	0,01 seg	10^{14} siglos
1.000	0,00001seg	0,001seg	1seg	Astronómico
10.000	0,000013 seg	0,01 seg	1, 7min	Astronómico
100.000	0,000017 seg	0,1 seg	2,8 horas	Astronómico

Figura 7.4.

El problema cuyo algoritmo tiene complejidad 2^n se convierte en intratable aún para pequeños tamaños de los datos. No le ocurre así al algoritmo n^2 -complejo, pero es evidente que para tamaños grandes puede requerir tiempo considerable.

La forma de crecimiento de los recursos necesarios en la ejecución de un determinado algoritmo se denota por $O(f(n))$, y ya hemos convenido anteriormente que sólo interesan los términos de mayor crecimiento de la función (el comportamiento asintótico).

Así las complejidades de la figura 7.4 son denotadas $O(\log_2 n)$, $O(n)$, $O(n^2)$ y $O(2^n)$. Un tiempo de ejecución de $14n^2 + 25n - 4$ se resume en que la complejidad es $O(n^2)$.

Los algoritmos cuyo comportamiento asintótico es del tipo $O(n)$, $O(n^2)$, $O(n^3)$, ..., en general $O(n^c)$ para c constante, se llaman *algoritmos polinómicos*, o de *complejidad polinómica*. Los algoritmos que se comportan como 2^n (en general c^n) son *algoritmos exponenciales*, o de *complejidad exponencial*.

Las figuras 7.4 y 7.5, examinadas conjuntamente, nos permiten apreciar que, siendo totalmente merecido el apelativo de "malos" con el que se conoce a los algoritmos exponenciales, también entre los polinómicos los hay mejores y peores y algunos pueden llegar a resultar bastante malos en cuanto que crece un poco el tamaño n .

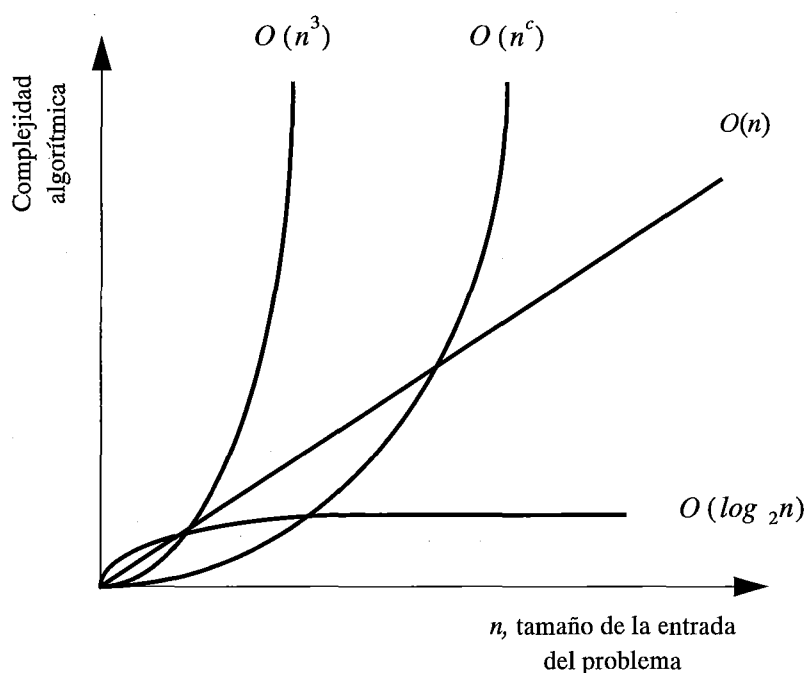


Figura 7.5.

3.2. Ejemplos de algoritmos con complejidad polinómica

Los algoritmos de búsqueda y clasificación (ordenación) constituyen una clase muy favorable para suministrar ejemplos didácticos acerca de la complejidad algorítmica, o de su inversa, la eficiencia del algoritmo.

En primer lugar, hacen fácil lo que muchas veces no lo es: precisar el tamaño de la entrada del problema, que en esta clase coincide con el número de elementos del conjunto sobre el que hay que operar la búsqueda o la ordenación. En segundo lugar y no menos importante, un algoritmo de búsqueda o de ordenación se traduce en un plan de operaciones básicas de comparación entre valores de elementos y de movimientos de estos elementos (en la ordenación), con la particularidad de que ambos tipos de operaciones dependen del número n de elementos del conjunto de partida.

Huelga decir que en esta categoría existen algoritmos muy sofisticados, en los que intervienen más operaciones que las puramente comparativas y de movimiento. Forman parte de las técnicas habituales en el campo de la gestión de ficheros y en general de los sistemas de gestión de bases de datos. No es nuestra intención entrar en ellos, aunque es justo dejar aquí señalado que todo su estudio se relaciona con problemas de eficiencia bajo diversas circunstancias.

3.2.1. Búsqueda secuencial

Consiste en explorar, consecutivamente y partiendo de un primero, los elementos de una lista o conjunto de n elementos comparando cada uno con una clave, que es el elemento buscado. Este elemento puede estar o no en el conjunto explorado.

El código siguiente en Pascal permite resolver esta búsqueda.

```
BEGIN
  encontrado: = False;
  ind: = 1;
  WHILE (NOT encontrado) AND (ind <= n) DO
    BEGIN
      IF lista [ind] = clave
      THEN encontrado: = True
      ELSE ind: = ind + 1
    END; {bucle de búsqueda}
  IF encontrado
  THEN BuscSec: = ind
  ELSE BuscSec: = 0
END; {Búsqueda secuencial}
```

(2)

La salida de este algoritmo, que llamamos BusSec, toma el valor 0 si no se encuentra la clave o el valor del lugar ocupado por el primer elemento coincidente con la clave, en caso positivo. "Encontrado" es una variable booleana e "ind" una variable de tipo intervalo formado por enteros. Los textos entre corchetes son comentarios. $< =$ y $> =$ significan menor o igual y mayor o igual, respectivamente.

El algoritmo codificado en la forma (2) no requiere explorar más de n elementos, en el peor de los casos. Y esto es prácticamente todo que hay que hacer.

Tanto la complejidad temporal como la espacial son $O(n)$. En adelante, sólo hablaremos de complejidad temporal.

3.2.2. Búsqueda binaria

El algoritmo de búsqueda binaria es mucho más eficiente (esto es, su complejidad mucho menor) que el de búsqueda secuencial. El algoritmo de búsqueda binaria tiene complejidad $O(\log_2 n)$. Véase la figura 7.5. Pero en realidad no son estrictamente comparables, porque el algoritmo de búsqueda secuencial es mucho más general, al no imponer condicionamiento alguno en cuanto al orden de los elementos en el conjunto. Por el contrario, en la búsqueda binaria se requiere previamente que el conjunto esté ordenado¹.

Este algoritmo trabaja comparando la clave con el elemento situado en medio de la lista. Si coincide, lo da como encontrado y si no, descarta la mitad (de ahí el nombre) de la lista que no puede contener la clave y repite el proceso.

De nuevo en Pascal, el algoritmo de búsqueda binaria en una lista de enteros puede expresarse en la forma de (3). Las variables "alto", "medio" y "bajo" son de tipo intervalo con enteros, como antes "ind".

Obsérvese que con cada comparación se divide la lista por la mitad, de modo que ésta se va convirtiendo en la mitad, en la cuarta, la octava... parte de su tamaño inicial. En el peor de los casos, el proceso continuará hasta que la lista a explorar esté vacía (en el programa, sentencia "ELSE BuscBin: =0"). En consecuencia, el mayor número de comparaciones que puede necesitar el método de búsqueda binaria será el primer valor entero k tal que $2^k \geq n$, es decir, $k \geq \log_2 n$.

```
BEGIN
  bajo: = 1;
  alto: = n;
  encontrado: = False;
  WHILE (NOT encontrado) AND (bajo < = alto) DO
    BEGIN
      medio: = (bajo + alto) DIV 2;
      IF lista [medio] = clave
      THEN encontrado: = True
      ELSE IF lista [medio] > clave
      THEN alto: = medio - 1
      ELSE bajo: = medio + 1
    END; {bucle de búsqueda}
  IF encontrado
  THEN BuscBin: = medio
```

¹El interés de aplicar la búsqueda binaria (precedida de un algoritmo de ordenación) frente a la búsqueda secuencial dependerá del número de veces que haya que aplicar el algoritmo de búsqueda, y, por supuesto, también del orden de complejidad del algoritmo de ordenación.

```
ELSE BuscBin: = 0
END; {búsqueda binaria}
```

3.2.3. Ordenación por el método burbuja

Ordenar es el proceso de reorganizar un conjunto de objetos según una determinada secuencia. Existen muchos métodos de ordenación y los más sencillos, denominados métodos directos, tienen complejidad $O(n^2)$. Normalmente, hay un número máximo de comparaciones y movimientos y un número mínimo, como resultante del grado inicial de ordenación del conjunto.

El método de la burbuja es un método de intercambio, que consiste en comparar dos elementos consecutivos, permutándolos cuando están desordenados y llegando hasta el final de la lista en cada una de las pasadas. Por ejemplo, si tenemos inicialmente la siguiente lista de números enteros (seguiremos utilizando por simplicidad este tipo de objetos):

(2, 4, 7, 6, 15, 12, 9, 10)

En una primera pasada (variable "Paso" en el código Pascal (4), la lista se reordenará así:

(2, 4, 6, 7, 12, 9, 10, 15)

```
BEGIN
  Intercambio: = TRUE;
  Paso: = 1;
  {El número de pasadas puede variar de 1
  hasta n - 1, dependiendo de la disposición inicial}
  WHILE (Paso <= n - 1) AND (Intercambio)
  DO BEGIN
    {Por el momento no se han producido
    intercambios}
    Intercambio: = False;
    FOR j: 1 TO n - Paso DO
      {si un elemento es menor que el anterior,
      hay que intercambiarlo}
      IF a [j] > a [j + 1]
      THEN BEGIN
        Intercambio: = TRUE;
        Aux: = a [j];
        a [j]: = a [j + 1];
        a [j + 1]: = Aux
      END; {if then}
      Paso: = Paso + 1
    END {while}
  END; {procedimiento burbuja}
```

(4)

La segunda pasada inicia los movimientos a partir de la pareja 12, 9, que es permutada, descendiendo 12 hasta el penúltimo lugar de la lista:

(2, 4, 6, 7, 9, 10, 12, 15)

El número de comparaciones en este algoritmo es $1/2 n(n-1)$ y los números mínimo, medio y máximo de movimientos son $0, 3/4(n^2 - n)$ y $3/2(n^2 - n)$, respectivamente. Por consiguiente, la complejidad es del orden $O(n^2)$.

3.3. Sinopsis

El esquema de la figura 7.6 pretende sintetizar algunas de las ideas ya vistas o que se desprenden de los conceptos y ejemplos examinados.

Un aspecto que no habíamos comentado anteriormente es el que se refiere, no a la complejidad del algoritmo, sino a la complejidad del problema. Tal cuestión remite a diseñar y comparar (en tiempo secuencial, por ejemplo) todos los algoritmos posibles para resolver un problema. En el apartado 2 mencionábamos distintas soluciones al problema de la multiplicación de dos números enteros de longitud n . Sus complejidades oscilaban entre $O(n^2)$ y $O(n \cdot \log_2 n \cdot \log_2 \log_2 n)$ o, lo que es lo mismo, entre $O(n^2)$ y $O(n \cdot \log n \cdot \log \log n)$.

Se conoce como cota superior de la complejidad de un problema a la complejidad del mejor algoritmo que se haya podido encontrar. En el caso de la multiplicación, la cota superior es $O(n \cdot \log n \cdot \log \log n)$, para recursos de computación secuencial. Es posible probar, a veces, que no existe algoritmo que pueda resolver un determinado problema sin emplear como mínimo una cierta cantidad de recurso, a la que se llama cota inferior. Si seguimos razonando acerca del problema de la multiplicación, convendremos en que no puede existir algoritmo más eficiente que el que necesitare un tiempo proporcional a n , puesto que siendo éste el tamaño de la entrada, es evidente que, como mínimo, cada dígito debería ser considerado al menos una vez.

¿Es posible encontrar algoritmo mejor que $O(n \cdot \log n \cdot \log \log n)$ para la multiplicación? ¿Puede probarse la inexistencia de un algoritmo más rápido? En suma, ¿cuál es la exacta complejidad de la multiplicación de dos números enteros? Estas son preguntas abiertas dentro de la teoría de la complejidad.

Otra observación. El análisis de algoritmos se concentra mucha veces en un algoritmo concreto, investigando en el interior de su orden de complejidad cual es la dinámica interna de ésta como resultado de las diversas configuraciones de datos. Esto conduce a evaluar medidas conocidas como casos peores, medios y mejores, de enorme interés en la práctica. El subapartado anterior nos ha ofrecido varios ejemplos en los que era patente una variabilidad de los tiempos de ejecución como producto de la configuración inicial de los datos. El lector puede imaginar hasta qué punto la dinámica de ejecución de predicados de un algoritmo complicado puede dificultar este tipo de análisis.

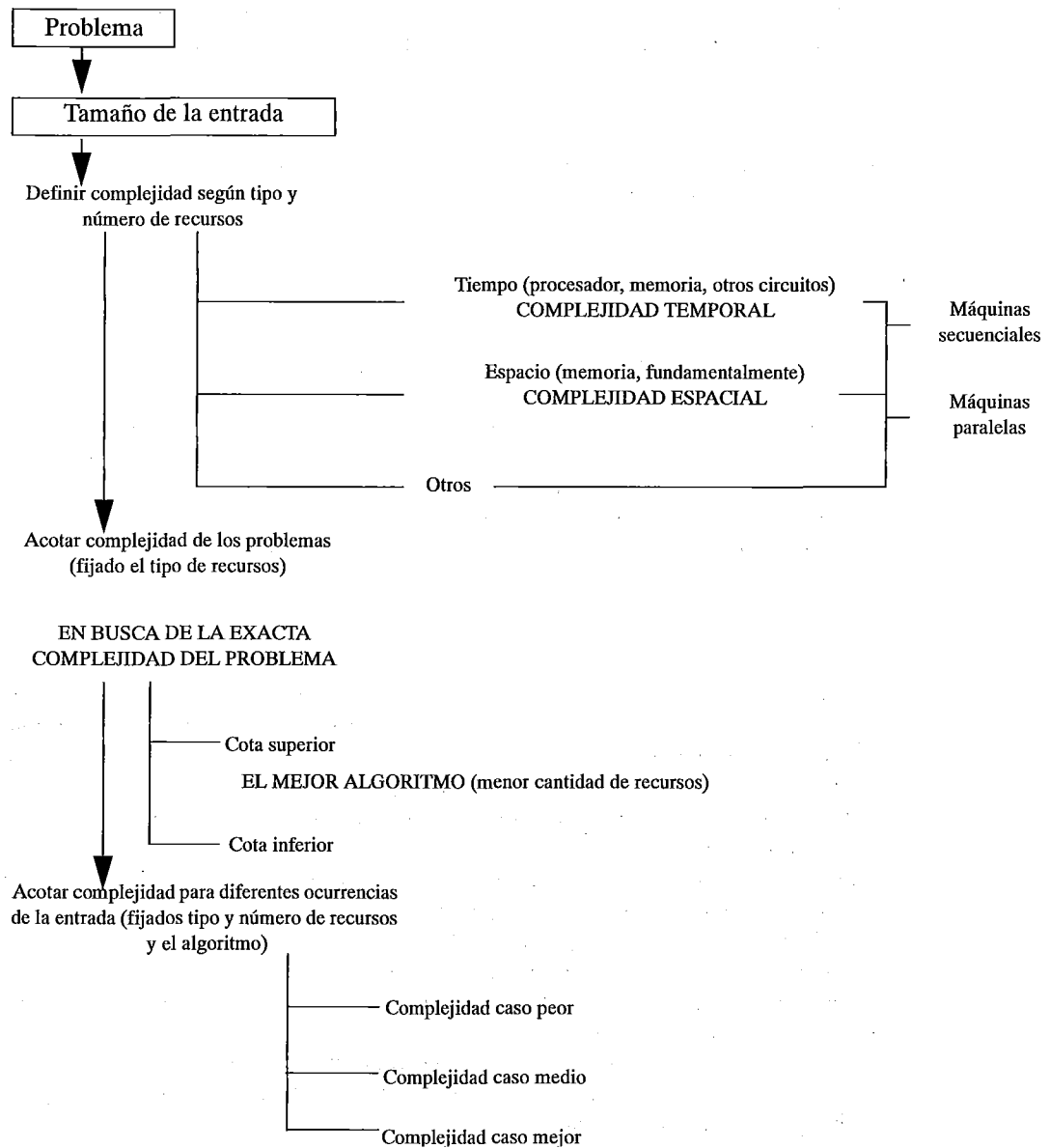


Figura 7.6.

Por último, un comentario sobre la aparente contradicción que nos sugiere la definición de complejidad algorítmica. Está muy claro que un algoritmo es tanto menos complejo cuantos menos recursos de máquina requiera. De ello venimos tratando a lo largo de todo el capítulo. En este sentido, es menos complejo el

algoritmo de ordenación Quicksort ($O(n \cdot \log n)$) que el de la burbuja. Sin embargo, el método de la burbuja se le ocurre a cualquiera, mientras que Quicksort lo inventó Hoare, precisamente uno de los más prestigiosos científicos de la informática. Desde una semántica antropológica, es más compleja (requiere más o por lo menos mejores recursos intelectuales) la solución Quicksort que la de la burbuja. En un caso estamos hablando de idear (inventar, diseñar, es un acto creativo) una solución y, en el otro, de ejecutarla mecánicamente.

4. Problemas P, NP y NP-completos

Hemos estado hablando de algoritmos más o menos eficientes, entendiendo que éstos son aquéllos cuyo crecimiento es polinómico. En general diremos que un algoritmo es eficiente si existe una máquina de Turing M que lo simule en tiempo $DTIME(n^i)$, donde i es una constante. A la clase de algoritmos eficientes se la denomina *la clase P*. Puede argumentarse, no sin razón, que un algoritmo (n^{50}) -limitado en tiempo, no resulta muy eficiente. Esto es cierto efectivamente, pero en la práctica casi todos los problemas de P tienen un grado pequeño.

4.1. El problema P-NP

Podría pensarse que con esta clase P ya tenemos definida la eficiencia, y sin embargo, nada más lejos de eso, pues existe otra clase de problemas denominados NP que vienen a perturbar toda la teoría de complejidad.

Como ya sugeríamos en el apartado 2.3, existen algoritmos (máquinas de Turing) llamados no-deterministas que, contrariamente a los algoritmos deterministas, no siguen un flujo fijo, sino que actúan en función de una serie de decisiones tomadas en tiempo real. *De entre los algoritmos no-deterministas existe un amplio conjunto de ellos que pueden considerarse eficientes, pero es inde-mostrable que estén en P* debido precisamente a que el algoritmo no es determinista. A esta clase de problemas se los llama NP . Dicho en palabras simples, la clase NP es el conjunto de problemas que tienen un algoritmo rápido (de complejidad polinómica) de verificación.

Un ejemplo de problema de tipo NP es el conocido problema de Hamilton. Dado un grafo como el de la figura 7.7, ¿puede encontrarse un camino que una todos los puntos y que pase una sola vez por cada punto? Este problema nos parece tener una solución determinista de tipo polinómico. Sin embargo, existen algoritmos no-deterministas que lo resuelven con eficiencia. Basta con elegir una posible solución al azar y verificar que cumple las condiciones del problema. Esta es precisamente una de las características más notables de los problemas de tipo NP . Encontrar el algoritmo determinista o la solución al problema es muy difícil, pero dada una posible solución, resulta sencillísimo comprobar que es válida. Esta diferencia es análoga a la que existe entre NP y P .

El problema principal en la teoría de los problemas intratables es, visto lo anterior, muy fácil de definir: ¿Es NP igual a P? *Este es el llamado problema P-NP*, y es tan difícil de resolver que casi nos vemos tentados a concluir que son dos clases diferentes. Lo que sí es demostrable es que todo problema de P es un problema de NP.

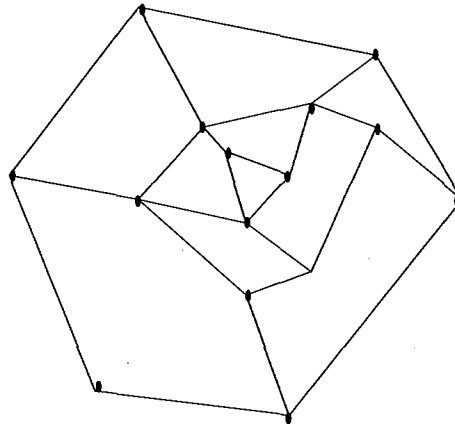


Figura 7.7.

Sin embargo, la cosa no acaba ahí, ya que existe una puerta abierta que permitiría demostrar que son iguales en el supuesto de encontrar la llave, si bien también podría demostrar que son distintos en caso de demostrar que la llave no existe. Vamos a ver esto con más detalle definiendo el concepto de completitud.

4.2. Completitud y problemas NP-completos

En la clase NP hay un cierto número de problemas que pueden catalogarse entre los más duros problemas, en el siguiente sentido: si se encontrase un algoritmo de tiempo polinómico para cualquiera de ellos habría un algoritmo de tiempo polinómico para todo problema en NP. Se dice que cualquier problema de esta categoría es *NP-completo*.

Un problema muy conocido entre los asiduos a la complejidad es el problema de "satisfactibilidad". Dada una expresión booleana, ¿es posible encontrar un conjunto de valores que den como resultado "true"?

El algoritmo NP es muy sencillo: cójase una posible solución al azar y verifique si el resultado es "true". Para demostrar que todo problema de NP puede reducirse al problema de satisfactibilidad (llamémoslo Lsat), hay que encontrar un algoritmo que permita simular cualquier otro problema de NP incrementando su complejidad por un factor que como mucho habrá de ser log. Este algoritmo existe pero es bastante complicado y no lo expondremos en detalle. El principio básico es construir una nueva máquina Lsat a partir de la máquina original M en

la cual las condiciones que deben verificarse para que M acepte la solución se convierten en una expresión booleana. De esta forma, la simulación de M con L_{sat} es como sigue: cójase una posible solución y verifíquese que la expresión booleana es "true". Si L_{sat} acepta la solución, M también acepta y viceversa, por lo que las dos máquinas son iguales. Si la complejidad de la máquina M era $T(n)$, la de la nueva máquina L_{sat} será como mucho $T(n) + \log(n)$.

Aparte del problema de Hamilton y el de satisfactibilidad hay otros problemas que también son NP-completos y son muy populares. Podemos mencionar unos cuantos:

- El problema del número cromático: Dado un número k , ¿existe una forma de dibujar un grafo con k colores de forma que dos vértices contiguos tengan un color diferente?
- El problema del agente de ventas: Dado un grafo, ¿cuál es el camino más corto que pasa una sola vez por cada uno de los puntos del grafo?
- El problema de las particiones: Dada una lista de enteros (i_1, i_2, \dots, i_k) , ¿existe algún subconjunto cuya suma sea exactamente $1/2(i_1 + i_2, \dots + i_k)$?

Entre toda la variedad de problemas NP-completos, hay algunos para los que se han realizado enormes esfuerzos en busca de un algoritmo determinista de tipo polinómico, sin encontrarlo. Puesto que todos o ninguno de estos problemas están en P, parece normal conjeturar que ninguno lo está. Casi parece más razonable buscar mejorar en los algoritmos heurísticos, que pueden resolverlos con bastante eficiencia en aplicaciones prácticas.

4.3. Un tema no cerrado

Podríamos seguir tratando otros muchos problemas dentro del tema de la complejidad, pero éstos son -valga la redundancia- muy complejos y requieren unos conocimientos de lógica más profundos.

Existen oráculos que intentan estudiar el qué pasaría si P fuese igual a NP, conjeturas relativas a la complementación de un problema y otras muchas técnicas que permitirían poder llegar a la solución del problema P-NP.

También existen desde luego problemas que no son ni de P ni de NP, es decir, que requieren tiempos exponenciales sin posibilidad de reducción. Este tipo de problemas, llamados "intrínsecamente difíciles" forman un conjunto no menos importante dentro de la densa jerarquía que compone la complejidad espacio-temporal.

5. Complejidad del software

Al final del subapartado 3.3, dedicado a una sinopsis de algunas de las ideas sobre complejidad algorítmica, se sugería una diferencia entre esta complejidad y la complejidad creativa específica del ser humano descubridor de algoritmos.

En informática, hay una tarea intermedia entre la de diseñar un algoritmo y la de ejecutarlo por un instrumento mecánico. Es la tarea de programar y codificar el algoritmo por medio de un lenguaje artificial. Aunque los lenguajes empleados cada día tienden a ser de mayor potencia expresiva (lo que quiere decir que una parte importante del trabajo de programación lo hace la propia máquina), esta actividad todavía es mayoritariamente de carácter humano. Por lo que sabemos, la codificación de un algoritmo puede introducir variaciones en su eficiencia de ejecución, pero no alterar el orden de magnitud de su complejidad. En cambio, determinadas circunstancias de esta tarea muestran una influencia profunda sobre el número de errores cometidos, la cantidad de esfuerzo requerido y en general sobre los costes del software. No existe una definición precisa de qué hay que entender por "complejidad del software", viniendo éste a ser un nombre genérico para sintetizar algunas de las mencionadas circunstancias, una especie de medida de cuán difícil es de comprender un programa y de trabajar con él.

La complejidad del software se traduce así, de una manera difusa, en un conjunto de recursos (en gran parte humanos) necesarios para producir software de calidad. Algún autor la llama "complejidad psicológica del software".

<i>Tipo</i>	<i>Medida literaria</i>	<i>Medida software</i>
1. Tamaño	Número de páginas de libro.	Número de instrucciones.
2. Dificultad de texto	Estilo de autor (por ejemplo, Borges versus Rulfo).	Operadores más operandos.
3. Estructural	Recurrencias y tramas entrelazadas.	Propiedades de grafos de estructura de control.
4. Intelectual	Tema (por ejemplo, mecánica cuántica versus plantas de interior).	Dificultad del algoritmo.

Figura 7.8.

No puede dejar de apuntarse un aspecto de la complejidad del software que se refiere al software de gran tamaño, cuya característica más sobresaliente es que ya no es un programa, sino tal vez cientos de ellos, formando conjuntos de cientos de miles o de millones de instrucciones imbricados en una operación común. Al desarrollo de este volumen de software se le llama *programación a gran escala*. A los problemas de complejidad ya evocados, es necesario añadir la complejidad de gestión simultánea de cientos de procesos humanos y técnicos, como los esquematizados en las figuras 1.1 y 4.2. Precisamente, la programación orientada a objetos resulta muy prometedora en cuanto a propiciar una considerable reducción del nivel de esta clase de complejidad.

Se han propuesto diversas métricas para estimar la complejidad de la programación. Terminaremos este capítulo sobre "complejidad" resumiendo la métrica conocida como Física (o Ciencia) del Software y la métrica del número ciclomático. El cuadro de la figura 7.8 resume, por analogía con medidas literarias, las clases de complejidad de las que aproximadamente estamos hablando en este apartado. Como se verá enseguida, la medida por medio de operadores y operandos se corresponde con la Física del Software y ciertas propiedades de grafos son el sustrato teórico del método del número ciclomático

5.1. Física del software

Esta métrica está basada en contabilizar los operandos y los operadores con que se codifica un determinado algoritmo en un lenguaje concreto. Los *operadores* son elementos sintácticos tales como +, -, >, IF THEN ELSE, END, etc... *Operandos* son las cantidades que reciben la acción de los operadores, variables y constantes, por ejemplo. En el programa en FORTRAN

```

      READ (5, 1) X
1     FORMAT (F 10.5)
      A = X/2
2     B = (X/A + A) /2
      C = B - A
      IF (C.LT.0) C = - C
      IF (C.LT.10.E - 6) GOTO 3
      A = B
      GOTO 2
3     WRITE (6, 1) B
      STOP
      END

```

(5)

transcrito en (5) podemos contabilizar lo siguiente:

- a) 13 operadores distintos. Al número de operadores distintos lo llamaremos n_1 (READ, FORMAT, =, /, (), +, -, .LT., IF, GOTO, WRITE, STOP, END).
- b) 24 operadores en total : N_1 (Cinco =, tres /, dos de los siguientes: (), -, .LT., IF, GOTO, y uno de los seis restantes).
- c) 11 operandos distintos: n_2 . (X, F10.5, A, 2, B, C, 10.E-6, 0, y las etiquetas 1, 2 y 3).
- d) 25 operandos en total: N_2 (Cinco de A y C, cuatro B, tres X, dos 2 y uno de los seis restantes).

A partir de estos elementos contables, la Física del software deriva funciones a las que denomina vocabulario y vocabulario potencial, nivel de programa, nivel de lenguaje, longitud de programa, volumen y volumen potencial del programa, contenido inteligente, esfuerzo de programación y otras, que presuntamente (éste

es un tema todavía abierto) permiten predecir el esfuerzo mental y el tiempo requeridos para codificar diferentes programas y para codificar un mismo algoritmo por medio de distintos lenguajes.

Número de operadores distintos	n_1
Número total de operadores	N_1
Número de operandos distintos	n_2
Número total de operandos	N_2
Tamaño del vocabulario	$n = n_1 + n_2$
Longitud del programa	$N = N_1 + N_2$
Volumen del programa	$V = N \log_2 n$
Vocabulario potencial	$n^* = 2 + n_2^*$
Volumen potencial	$V^* = n^* \log_2 n^*$
Longitud calculada del programa	$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
Nivel del programa	$L = V^*/V$
Nivel calculado del programa	$\hat{L} = 2n_2/n_1N_2$
Nivel del lenguaje	$\lambda = LV^*$
Contenido inteligente	$I \approx V^*$
Esfuerzo	$E = V/L$

Figura 7.9.

El volumen del programa (figura 7.9) se mide en bits. El nivel del programa L representa una medida de lo sucinta que puede ser la codificación de un programa. Cuanto más elevado sea el nivel del lenguaje utilizado en el programa, tanto más se aproxima la implementación a una llamada de procedimiento. Recordemos al respecto la sentencia CALL MAX (A, B, MCD) frente al código del mismo programa en PL/I del apartado 2 del capítulo 1. Por definición $L=V^*/V$, donde V^* es el volumen del algoritmo codificado por una sentencia de llamada CALL (n_2^* suele representar el número de parámetros diferentes de entrada y salida; en el caso del algoritmo del m.c.d. $n_2^*=3$) y $0 < L \leq 1$, siendo L tanto más próximo al valor 1 cuanto más potente es el lenguaje empleado. El algoritmo (5) se podría codificar con CALL RAIZCUA (X, B), con $n_2^*=2$, siendo B la raíz cuadrada de

X , \hat{N} y \hat{L} son fórmulas predictor de la longitud y del nivel del programa, respectivamente.

Llama la atención la expresión empleada para el esfuerzo E que, como se puede observar, ha sido interpretado como el número total de discriminaciones mentales requeridas para codificar un determinado programa. Su valor, que podría tomarse como el equivalente a una medida de la complejidad de programación es tanto mayor cuanto mayores son el tamaño del vocabulario y la longitud del programa (ambos se traducen en el valor de V) y más próximo a la máquina el lenguaje empleado. Diversos estudios empíricos demuestran la existencia de una fuerte correlación entre E y el tiempo necesario para la programación, y asimismo entre E y el número de errores hallados en los programas.

En cuanto al contenido de inteligencia I del programa, esta función tiende a ser constante e independiente del nivel de codificación.

Si aplicamos el cuadro anterior de definiciones al programa (5) obtenemos los valores de la figura 7.10.

$$n_1 = 13$$

$$N_1 = 24$$

$$n_2 = 11$$

$$N_2 = 25$$

$$n = 24$$

$$N = 49$$

$$V = 49 \cdot \log_2 24 = 224,66$$

$$n^* = 2 + 2 = 4$$

$$V^* = 4 \log_2 4 = 8$$

$$\hat{N} = 13 \log_2 13 + 11 \log_2 11 = 86,16$$

$$L = 4 \log_2 4 / 49 \log_2 24 = 0,0356$$

$$\hat{L} = 2 \cdot 11 / 13 \cdot 25 = 0,0676$$

$$\lambda = (4 \log_2 4 / 49 \log_2 24) \cdot 4 \log_2 4 = 0,2848$$

$$I \approx 4 \log_2 4 = 8$$

$$E = 49 \log_2 24 \div 4 \log_2 4 / 49 \log_2 24 = 6.310,67$$

Figura 7.10.

5.2. Número ciclomático

La métrica que brevemente describiremos ahora se basa en la estructura decisional (flujo de predicados) de los programas. A esta estructura se le asocia un grafo orientado G . Recuérdese la figura 1.4 y las correspondencias allí establecidas entre bloques de código y ramas de programa con nodos y arcos.

Por la teoría de grafos, se calcula el número de circuitos de control linealmente independientes en G . Un grafo fuertemente conectado es aquel para el que existe un camino entre cualquier pareja de nodos.

El número ciclomático $V(G)$ de un grafo con n nodos, e arcos y p componentes conectado es:

$$V(G) = e - n + 2p \quad (6)$$

Existe un teorema que dice lo siguiente: en un grafo G fuertemente conectado, el número ciclomático es igual al número máximo de circuitos linealmente independientes.

Así pues, el número ciclomático da una medida de la complejidad del software, entendida como dificultad mental en la tarea de programación.

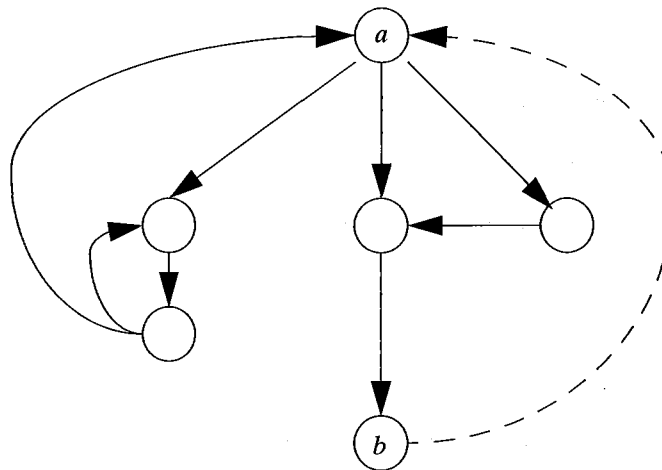
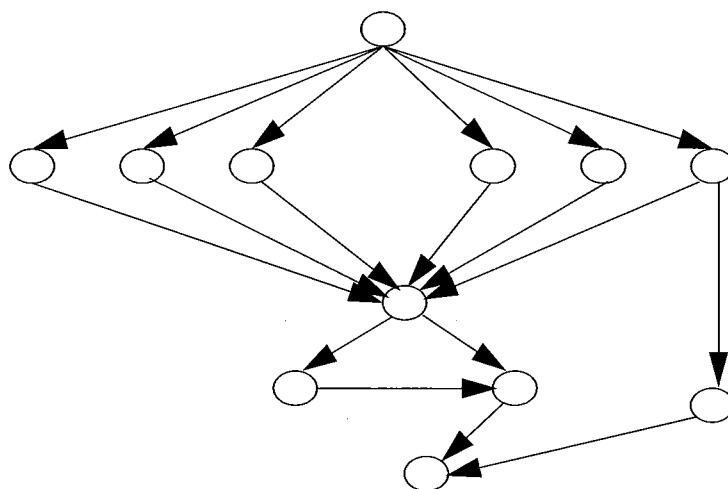


Figura 7.11.

El grafo de la figura 7.11 corresponde a un programa cualquiera, cuyos nodos de entrada y de salida son a y b . Si se cierra de forma ficticia por el arco $b-a$, el grafo resulta ser fuertemente conectado. Su número ciclomático es 5 ($e = 9$, $n = 6$, $p = 1$). El grafo de la figura 1.4 tiene un ciclomático de 4. El grafo de la figura 7.12, tiene un ciclomático de 8.

6. Resumen

Existe un conjunto bastante grande de algoritmos demasiado difíciles de llevar a la práctica por limitaciones de tiempo o espacio.

**Figura 7.12.**

La complejidad de un algoritmo no depende de la velocidad del ordenador, de su capacidad de memoria o formato de instrucciones, es decir, es independiente del hardware.

Existen numerosas clases de complejidad según sean los algoritmos deterministas -los pasos del algoritmo son siempre los mismos- o no deterministas -se toman decisiones que pueden variar la cantidad de pasos ejecutados.

Los algoritmos que pueden ejecutarse en un tiempo de orden polinómico en función de la magnitud de entrada, se consideran eficientes y constituyen la clase P, mientras que aquéllos que requieren algoritmos de tipo exponencial se consideran difíciles o complejos.

Algunos problemas pueden resolverse eficientemente con algoritmos no deterministas pero su solución determinista es compleja. A este tipo de algoritmo se le llama NP. En ciertos casos, un algoritmo NP es tal que todo problema NP puede simularse con él, en cuyo caso decimos que el algoritmo es NP-completo.

Si lográramos demostrar que un algoritmo NP-completo puede simularse mediante un algoritmo determinista eficiente -es decir, de P-, se demostraría automáticamente que todos los problemas NP pueden simularse con un algoritmo determinista eficiente.

Los estudios realizados hasta el momento no logran, sin embargo, aclarar este dilema llamado "problema P-NP", siendo una tendencia actual la de considerar

que la clase NP es diferente de la clase P. De esta forma pueden centrarse los estudios en la búsqueda de métodos no deterministas para resolver óptimamente toda esta gama de problemas.

En paralelo con la complejidad algorítmica surge una preocupación por la complejidad del software, correspondiente a una rama mucho menos teórica de la informática, conocida hoy por el nombre de ingeniería del software.

En este capítulo se han presentado dos clases de métrica, la primera de carácter lingüístico basada en la contabilidad de los operadores y operandos utilizados en la codificación de los programas, y la segunda, en la evaluación del grafo asociado a la estructura de control de los programas. Cualquiera de estas dos clases de técnicas y otras muchas que se han elaborado sólo son aplicables en el ámbito de los lenguajes clásicos de tipo imperativo y de los ordenadores de funciona-miento secuencial según el modelo de von Neumann.

Tanto el campo de la complejidad algorítmica como el de la complejidad del software siguen abiertos y muy activos.

7. Notas histórica y bibliográfica

El estudio de la complejidad algorítmica puede considerarse bastante reciente. Parece que el concepto de complejidad temporal se origina hacia 1965 en un artículo de Hartmanis y Stearns. La complejidad espacial se debe a Hartmanis, Lewis y Stearns, por un trabajo presentado a un simposio de Teoría y Diseño de circuitos lógicos en 1965, en el que estos autores analizaban la limitación de los cálculos por causa de la capacidad de memoria (Hopcroft y Ullman, 1979).

El número de investigadores y trabajos ha crecido y se ha diversificado sostenidamente desde entonces, por lo que su simple mención alargaría esta nota sin mayor provecho.

Sin embargo, no puede dejarse de citar a Cook, por su formulación de la clase de problemas NP-completos y del problema P-NP. Esto sucedió en 1971. Al año siguiente y sucesivos, Karp y otros matemáticos enunciaron listas de problemas NP-completos, entre los que se alinean todos los presentados en el subapartado 4.2.

Es Karp precisamente quien, en su conferencia de aceptación del premio Turing (Karp, 1986), ha expuesto una panorámica histórica de algunos de los hitos en la teoría de complejidad algorítmica, que merece ser leída. Un aliciente mayor adicional para aquellos lectores que deseen conocer un auténtico marco conceptual de los problemas planteados a la Informática Teórica en el eje computabilidad-complejidad lo constituye un rompecabezas resuelto por Karp, cuyas piezas son los mencionados problemas. Lo publica el mismo número de la revista *Communications of the A.C.M.*, que recoge la citada conferencia (Frenkel, 1986).

Díaz (1982) y Balcázar, en (Gamella, 1985, pp. 61-65) describen algunos de los campos a los que se extienden en la actualidad los trabajos relativos a la complejidad algorítmica. Citando a este último, nos encontramos con parcelas dedicadas:

1. Al desarrollo de técnicas de análisis mediante funciones continuas interpoladoras del comportamiento asintótico de algoritmos dados.
2. Al diseño de estructuras de datos eficientes, analizando el tiempo requerido por los distintos algoritmos.
3. Al planteamiento de problemas de investigación operativa (tales como búsqueda de rutas mínimas que permitan recorrer diversos puntos y otras funciones de coste) en términos combinatorios.
4. Al análisis de comportamientos "caso medio".
5. Al análisis de algoritmos para problemas de tipo geométrico, frecuentes en las áreas de robótica (algoritmos para decidir formas y movimientos) y del diseño de circuitos integrados a gran escala (algoritmos para integrar en una misma pastilla de silicio una enorme cantidad de componentes y conectarlos adecuadamente con el mínimo posible de interconexiones cableadas).

En la parte de complejidad algorítmica, este capítulo ha sido confeccionado con los materiales siguientes.

Las grandes líneas del capítulo se han inspirado en una mezcla de los libros de Goldschlager y Lister (1982) y de Hopcroft y Ullman (1979), ayudada por un trabajo de síntesis, hoy día inencontrable, de uno de nuestros alumnos, Díez Medrano (1984). El primero de estos libros es una introducción a la informática y presenta el tema de una manera mucho más intuitiva que el segundo, escrito en un nivel alto y considerablemente formalizado.

Para las definiciones de complejidad espacial y temporal en máquinas de Turing hemos seguido muy de cerca a Hopcroft y Ullmann en su capítulo 12. También, los resultados del subapartado 2.4 han sido extractos de este libro. En gran parte, asimismo el apartado 4 es tributario de dichos autores a través de la versión de Díez Medrano.

Los ejemplos de algoritmos de búsqueda y ordenación se encuentran, con todos los detalles de codificación (declaración de variables y demás especificaciones) y otros muchos ejemplos, en un texto didáctico sobre algoritmos y complejidad, orientado a la enseñanza de la programación debido a Garijo y otros autores (Garijo *et al.*, 1985). Aunque este texto de apuntes es prácticamente inac-

cesible en el tiempo de la reedición del presente libro, no podemos dejar de rendirle la debida referencia.

Por lo que respecta a la complejidad del software, los trabajos se inician aún más recientemente. Si no estamos equivocados, fue hacia 1972 cuando Halstead comenzó a publicar, bajo forma de informes técnicos en la Universidad norteamericana de Purdue, unas primeras aportaciones que, primero bajo los nombres de "Termodinámica de algoritmos", "Leyes naturales que controlan la estructura de los algoritmos" y otros parecidos, derivaron después en "Física del software" y finalmente, en "Ciencia del software". Halstead se inspiró en investigaciones sobre lenguajes naturales de Zipf en 1949 (seguidas y formalizadas posteriormente por Mandelbrot) y de Shannon sobre teoría matemática de la información.

La aplicación del número ciclomático a la complejidad de programas data de 1976 y se debe a Th. McCabe.

Una descripción panorámica de varias de estas métricas de complejidad del software y algunas referencias más se encuentran en Sáez Vacas, (1992, 1994).

El cuadro de la figura 7.8 puede encontrarse, junto a las propuestas de Zipf y Mandelbrot en Shooman (1983, cap. 3).

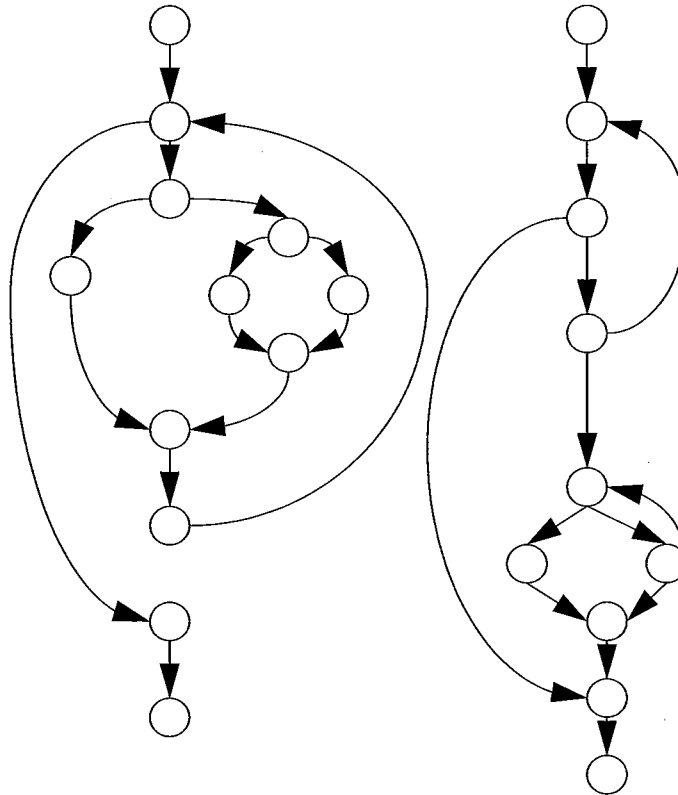
Para la presentación de los postulados de la métrica de Halstead hemos utilizado el ejemplo (5), reproducido del capítulo 2 del libro de Shooman (1983), algunos elementos del artículo de Coulter (1983) y el libro en el que finalmente Halstead resumió su teoría (Halstead, 1977).

En cuanto al número ciclomático, nos hemos valido del artículo original de su propio autor (McCabe, 1976).

8. Ejercicios

- 8.1. Buscar en la bibliografía sobre algoritmos de búsqueda y clasificación una selección de éstos, familiarizarse con ellos y con su complejidad, comparativamente a los vistos en este capítulo.
- 8.2. Aplicar la métrica de los operandos y los operadores a los códigos (2), (3) y (4) escritos en Pascal.
- 8.3. Tomar cualquiera de los algoritmos representados en (2), (3) y (4), codificarlo en un lenguaje ensamblador. Aplicar al código obtenido la métrica de Halstead y comparar los resultados con los resultados correspondientes al código en Pascal.
- 8.4. Aplicar la métrica del número ciclomático a los mismos algoritmos (2), (3) y (4).

- 8.5.** Aplicar la métrica del número ciclomático a los siguientes grafos asociados a sendos programas.



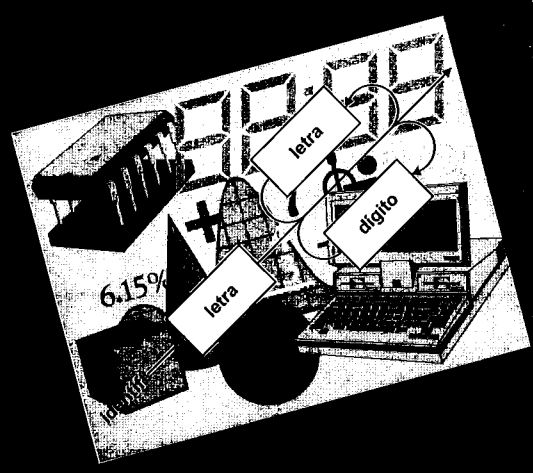
- 8.6.** Aplicar la métrica del número ciclomático a los programas de las figuras 4.3 y 4.5, comparar y extraer alguna conclusión.

Parte IV

Lenguajes

Fundamentos de informática

1



1

Ideas Generales

1. Lenguajes e informática

Ya al comienzo de este libro (tema "Lógica", capítulo 1, apartado 4) hemos definido informalmente un lenguaje como un sistema de símbolos que permite la comunicación entre personas, entre personas y máquinas o entre máquinas. Hemos introducido también (en el apartado 5, que debería releerse ahora) la terminología y los primeros conceptos de la teoría de lenguajes formales.

La importancia de la teoría de lenguajes en informática radica en la correspondencia biunívoca que existe entre máquinas programables y lenguajes. A cada máquina corresponde un lenguaje en el que se escriben sus programas; a la inversa, a cada lenguaje de programación le corresponde una máquina que interpreta los programas escritos con él¹. La teoría de lenguajes permite desarrollar de manera científica tanto la creación y la producción de programas como el diseño de máquinas y lenguajes de programación.

Los tres capítulos centrales del tema que ahora comenzamos están dedicados a presentar los conceptos básicos de la teoría de lenguajes formales y su relación con la teoría de autómatas. Después, en el último capítulo, veremos la utilidad de estos conceptos en los lenguajes de programación de ordenadores.

¹ El término "máquina" debe entenderse aquí en un sentido abstracto. Por ejemplo, una "máquina BASIC" es una "máquina virtual" cuyo lenguaje de máquina es el BASIC y que se materializa (se "implementa") mediante una máquina real (un ordenador) y unos programas adecuados.

2. Descripciones de los lenguajes

La descripción formal de un lenguaje finito es inmediata: basta enumerar las cadenas que lo forman. Pero si es infinito, será preciso inventar una descripción finita; esta descripción será, a su vez, una cadena de símbolos combinados de acuerdo con ciertas reglas (sintaxis) y con un determinado significado, tanto para los símbolos como para las reglas (semántica). Por tanto, esta cadena de símbolos pertenecerá a un *metalenguaje*, que servirá para describir nuestro lenguaje. Por ejemplo, una expresión regular (tema "Autómatas", capítulo 4) es una cadena del lenguaje de las expresiones regulares, que es un metalenguaje para describir los lenguajes regulares.

Ahora bien, podemos preguntarnos si, dado un lenguaje finito cualquiera, $L \subset A^*$, es posible siempre describirlo de manera finita. La respuesta es "no". En efecto, sabemos (tema "Algoritmos", capítulo 6, apartado 3.1) que A^* (conjunto de todas las cadenas construidas sobre un alfabeto A) es recursivamente numerable (a cada cadena se le puede asociar, por ejemplo, su número de Gödel), y, por tanto, cualquier $L \subset A^*$ también lo es. Supongamos que existe un metalenguaje $L_M \subset A_M^*$ tal que cada cadena de L_M es una representación finita de un lenguaje sobre A^* , es decir, tal que a cada $L(M) \subset A^*$, corresponde al menos un $x_M \in L_M$. Si tal metalenguaje existiera debería ser, como todo lenguaje, recursivamente numerable. Esto implicaría que el conjunto de todos los $L \subset A^*$ fuera recursivamente numerable. Sin embargo, existe un teorema de la teoría de conjuntos según el cual el conjunto de los subconjuntos de un conjunto recursivamente numerable (y A^* lo es) no es recursivamente numerable, lo que contradice la conclusión anterior y refuta la hipótesis de que exista un L_M para todo $L \subset A^*$. Por consiguiente, podemos concluir que *no todos los posibles lenguajes tienen una representación finita en un metalenguaje*.

En los dos capítulos siguientes estudiaremos la descripción de ciertos tipos de lenguajes desde en punto de vista *generativo*: una *gramática*, G , es una descripción metalingüística con la que se puede desarrollar un algoritmo enumerativo para generar las cadenas del lenguaje $L(G)$, es decir, se puede diseñar una máquina de Turing que haga corresponder el número 1 a la cadena x_1 , el 2 a la x_2 , etc., de manera que genere el conjunto recursivamente numerable $L(G) = \{x_1, x_2, \dots\}$.

En el capítulo 4 consideraremos un punto de vista complementario: el del *reconocimiento*, o sea, el estudio de algoritmos o estructuras de máquinas que permiten, dado un lenguaje L y una cadena x , determinar si $x \in L$ o $x \notin L$.

Finalmente, en el capítulo 5 abordaremos el aspecto *interpretativo*: un programa se escribe con el objetivo de que la máquina a la que va destinado *ejecute* un determinado algoritmo. Para que ello sea posible es necesario que el programador sepa con exactitud cómo la máquina va a reaccionar frente a cada una de las instrucciones de su programa, es decir, qué *significan* (y entramos así en el campo de la semántica) para la máquina las distintas cadenas del lenguaje.

3. Sintaxis, semántica y pragmática.

Actualmente, en la teoría de lenguajes formales, el campo de la sintaxis está bien establecido: como veremos en el capítulo siguiente, existen metalenguajes (gramáticas) formales que describen con precisión varios tipos de lenguajes y que resultan de gran utilidad práctica en informática para el diseño de lenguajes de programación y para el reconocimiento, traducción e interpretación de programas.

La semántica, hasta ahora, ha resultado más difícil de formalizar. En la definición de casi todos los lenguajes de programación, la semántica se expresa de manera informal, mediante explicaciones verbales en sus manuales sobre lo que significa cada posible sentencia. La formalización de la semántica es, sin embargo, importante desde un punto de vista práctico. Con una definición formal de la semántica de un lenguaje se podrían elaborar métodos sistemáticos de comprobación de programas, es decir, de verificar, antes de su ejecución, que los programas son correctos, en el sentido de que van a hacer lo que se pretende que hagan Y, eventualmente, se podrían diseñar herramientas (es decir, otros programas) basadas en estos métodos para automatizar la verificación de programas. Ello contribuiría a mejorar la productividad de los procesos de producción de software y la fiabilidad de los programas. Se han propuesto varios enfoques para definir formalmente la semántica de los lenguajes de programación (los comentaremos en el capítulo 5), pero éste es todavía un asunto sometido a numerosos trabajos de investigación.

En algunos textos sobre lenguajes de programación se habla de "pragmática", pero aún estamos lejos, no ya de disponer de una formalización, sino siquiera de estar de acuerdo sobre lo que representa: para unos, se refiere a los aspectos de comunicación con el usuario, para otros atañe a los problemas que se presentan cuando se pasa de la definición formal de un lenguaje a las particularidades de las máquinas en las que los programas han de ejecutarse, según otros trataría de los "aspectos prácticos" de la actividad de programar.

4. Dos ejemplos

Los ejemplos que desarrollamos a continuación nos servirán para introducir de una manera intuitiva algunos de los conceptos que formalizaremos en capítulos posteriores.

Puesto que aún no hemos definido lo que es una gramática, haremos uso de una que, más o menos vagamente, todos conocemos: la gramática del lenguaje castellano.

Para describir la formación de sentencias (oraciones) utilizaremos *categorías sintácticas o sintagmas*, como "nombre", "verbo", etc. (o sintagma nominal, sintagma verbal, etc.). Está claro que, si bien "nombre" es un nombre,

"verbo" no es un verbo. Esto es debido a que para describir el lenguaje castellano utilizaremos como metalenguaje el propio castellano, y ponemos las comillas para destacar que la palabra en cuestión se utiliza como elemento del metalenguaje (es la diferencia entre *uso* y *menção* del lenguaje, ver tema "Lógica", capítulo 1, apartado 5.3). La notación habitualmente seguida no es poner comillas, sino encerrar la palabra entre paréntesis angulares: ⟨nombre⟩, ⟨verbo⟩, etc.

Una sentencia castellana sintácticamente correcta estará compuesta por palabras concretas pertenecientes a las diversas categorías sintácticas (uno o varios sintagmas nominales, uno o varios sintagmas predicativos, etc.) combinadas de acuerdo con ciertas reglas.

Por ejemplo, una regla puede ser:

(1) *Para formar una sentencia, póngase un sintagma nominal y a continuación un sintagma predicativo.*

Y otras:

(2) *Un nombre propio es un sintagma nominal.*

(3) *"España" es un nombre propio.*

(4) *Un sintagma predicativo puede formarse con una forma verbal transitiva y un sintagma nominal.*

(5) *"es" es una forma verbal transitiva.*

(6) *Un sintagma nominal puede formarse con un determinante y un nombre común.*

(7) *Un artículo es un determinante.*

(8) *"un" es un artículo.*

(9) *"estado" es un nombre común.*

Del conjunto de estas nueve reglas puede deducirse que

España es un estado

es una sentencia del castellano.

Las reglas anteriores pueden expresarse formalmente así²:

(1) ⟨Sentencia⟩ → ⟨SN⟩ ⟨SP⟩

(2) ⟨SN⟩ → ⟨Npr⟩

² El símbolo "→" no tiene ahora nada que ver con el condicional de la lógica. En este contexto significa: "la parte de la izquierda puede sustituirse por la parte de la derecha", o "la parte izquierda puede tener la forma de la parte derecha".

- (3) <Npr> → España
- (4) <SP> → <FVtr> <SN>
- (5) <FVtr> → es
- (6) <SN> → <Det> <Ncom>
- (7) <Det> → <Art>
- (8) <Art> → un
- (9) <Ncom> → estado.

Y la sentencia obtenida puede descomponerse sintácticamente de acuerdo con las reglas; esta descomposición puede indicarse mediante corchetes etiquetados:

[[España]] [[es] [[un]] [estado]]]
 SN Npr SP FVtr SN Det Art Ncom

o bien, de una forma más gráfica, con un árbol llamado *árbol de derivación* o *árbol sintáctico* (figura 1.1).

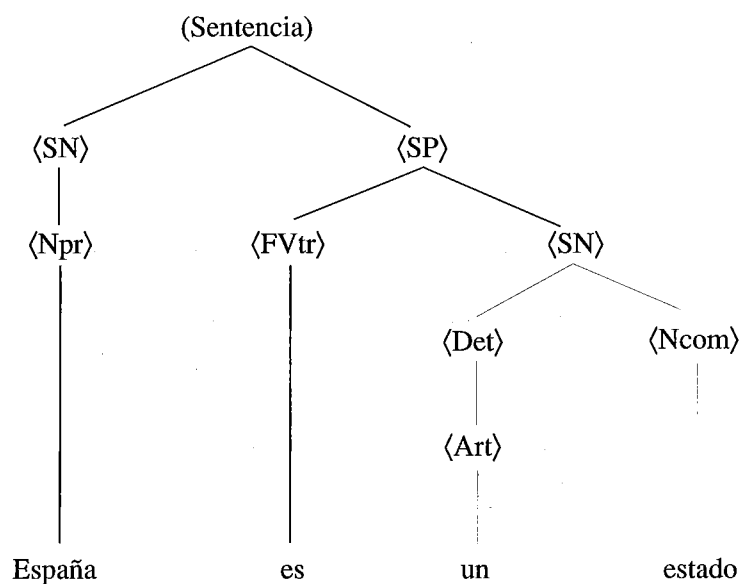


Figura 1.1.

Como segundo ejemplo, consideraremos la sentencia

La soberanía nacional reside en el pueblo español.

Su descomposición sintáctica, o derivación a partir de las reglas, es la indicada por el árbol de la figura 1.2.

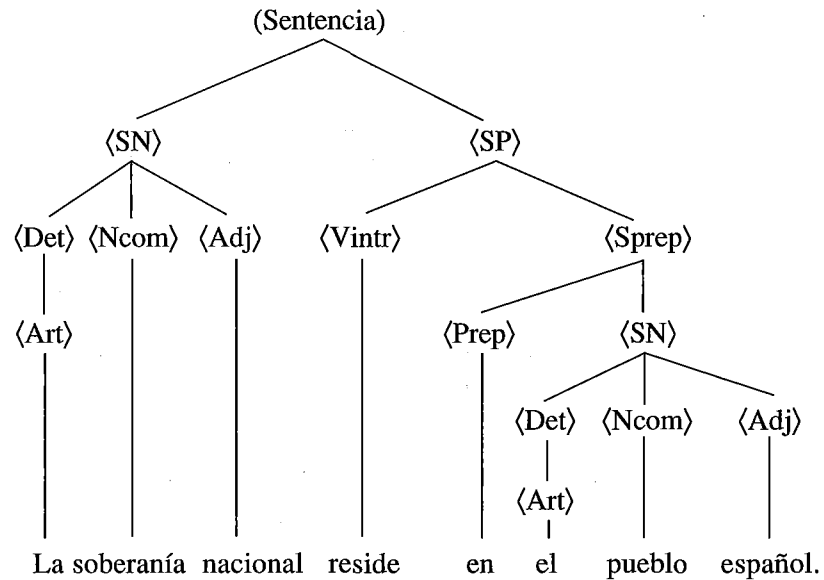


Figura 1.2.

Las reglas utilizadas para esta derivación han sido:

- (1) <Sentencia> → <SN> <SP>
- (2) <SN> → <Det> <Ncom> <Adj>
- (3) <Det> → <Art>
- (4) <SP> → <Vintr> <Sprep>
- (5) <Sprep> → <Prep> <SN>
- (6) <Art> → la
- (7) <Art> → el
- (8) <Ncom> → soberanía
- (9) <Ncom> → pueblo
- (10) <Adj> → nacional
- (11) <Adj> → español
- (12) <Vintr> → reside
- (13) <Prep> → en

Se obtiene una sentencia partiendo de la regla (1) y aplicando las demás hasta que resulta una cadena con sólo *símbolos terminales*, en este caso, las palabras concretas del castellano obtenidas por aplicación de las *reglas terminales* (6) a (13).

Con estas reglas pueden derivarse otras sentencias, como

"El pueblo español reside en la soberanía nacional",

"La pueblo nacional residen en la pueblo español".

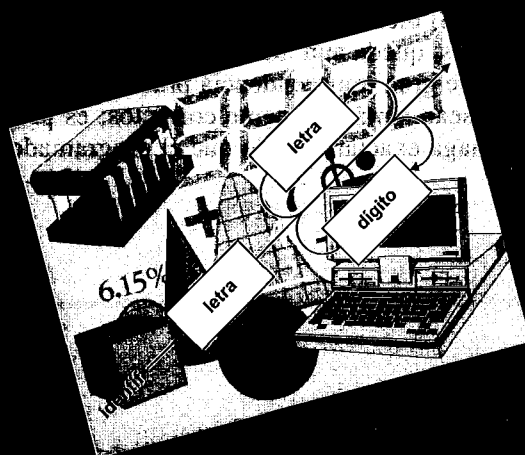
etc., que serían sentencias correctas en la gramática definida por tales reglas.

5. Resumen

La teoría de lenguajes formales es una herramienta matemática que permite abordar con rigor el diseño de lenguajes de programación y la producción de software. Si bien existe ya una buena elaboración de la misma en el campo de la sintaxis, queda aún mucho por hacer en cuanto a la semántica, y éste es un aspecto de gran transcendencia práctica y económica: no basta con que los programas sean sintácticamente correctos; es preciso que la máquina que los interpreta haga exactamente lo que el programador pretende que haga.

Fundamentos de informática

2



Gramáticas y lenguajes

1. Definición de gramática

Una gramática es una cuádrupla

$$G = \langle E_T, E_A, P, S \rangle,$$

donde

E_T es un conjunto finito de símbolos llamado *alfabeto principal o alfabeto de símbolos terminales*¹.

E_A es un conjunto finito de símbolos llamado *alfabeto auxiliar o alfabeto de variables*.

P es un conjunto finito de pares ordenados (α, β) , donde $\alpha \in (E_T \cup E_A)^+$ y $\beta \in (E_T \cup E_A)^*$. Estos pares se llaman *reglas de escritura o producciones*, y generalmente se escriben con la notación $\alpha \rightarrow \beta$. α es el *antecedente* de la regla y β el *consecuente*. Si $\beta \in E_T^*$ se dice de la regla que es una *regla terminal*.

¹ En adelante representaremos con "E" a un alfabeto, porque "A" se utilizará frecuentemente como uno de los símbolos del alfabeto auxiliar.

$S \in E_A$ es un símbolo destacado del alfabeto auxiliar llamado *símbolo inicial*.

Llamaremos $E = E_T \cup E_A$; supondremos que $E_T \cap E_A = \emptyset$, y para distinguir los elementos de los diferentes conjuntos adoptaremos el convenio de utilizar:

- a) letras mayúsculas del alfabeto latino para los elementos de E_A . (O bien, palabras del metalenguaje castellano entre paréntesis angulares);
- b) letras minúsculas del comienzo del alfabeto latino (a, b, c, \dots), o cifras, para los elementos de E_T . (O bien, palabras del castellano);
- c) letras minúsculas del final del alfabeto latino (w, x, y, z) para los elementos de E_T^* ;
- d) letras minúsculas del alfabeto griego para los elementos de E^+ .

2. Relaciones entre cadenas de E^*

2.1 Relación de derivación directa, \Rightarrow_G

Si $(\alpha \rightarrow \beta) \in P$ y $\gamma, \delta \in E^*$, las cadenas $\gamma\alpha\delta$ y $\gamma\beta\delta$ están en la *relación de derivación directa* en la gramática G . Escribiremos entonces

$$\gamma\alpha\delta \Rightarrow_G \gamma\beta\delta$$

y diremos que la cadena $\gamma\beta\delta$ *deriva directamente* de la $\gamma\alpha\delta$ o bien que $\gamma\alpha\delta$ *produce directamente* $\gamma\beta\delta$ en la gramática G . (De ahí el nombre de producciones para los elementos de P .)

2.2 Relación de derivación, $\xRightarrow{*}_G$

Dados $\alpha_1, \alpha_m \in E^*$, diremos que están en la *relación de derivación* en la gramática G si existen $\alpha_2, \alpha_3 \dots \alpha_{m-1}$ tales que

$$\alpha_1 \Rightarrow_G \alpha_2; \alpha_2 \Rightarrow_G \alpha_3; \dots \alpha_{m-1} \Rightarrow_G \alpha_m$$

Se escribirá entonces

$$\alpha_1 \xRightarrow{*}_G \alpha_m$$

diciendo que α_m deriva de α_1 , o que α_1 produce α_m

Por convenio, $\alpha \xrightarrow[G]{*} \alpha \quad \forall \alpha \in E^*$

Siempre que sea evidente que nos referimos a una determinada gramática, G , escribiremos \Rightarrow y $\xrightarrow[G]{*}$ en lugar de \Rightarrow_G y $\xrightarrow[G]{*}$

Obsérvese que ni \Rightarrow ni $\xrightarrow[G]{*}$ son relaciones de equivalencia, ya que, en general, no son simétricas.

3. Lenguaje generado por una gramática. Equivalencia de gramáticas

Una cadena $\xi \in E^*$ es una *forma sentencial* de la gramática G si existe una derivación que produce ξ a partir de S , es decir, si $S \xrightarrow[G]{*} \xi$.

Si además ξ sólo contiene símbolos terminales entonces es una *sentencia* o cadena válida. El conjunto de sentencias que puede generarse en una gramática G se llama *lenguaje generado por la gramática G* , es decir:

$$L(G) = \{x \mid x \in E_T^* \text{ y } S \xrightarrow[G]{*} x\}$$

Dos gramáticas, G_1 y G_2 , son equivalentes si ambas generan el mismo lenguaje: $L(G_1) = L(G_2)$.

4. Ejemplos

Ejemplo 4.1

Sea la gramática definida por $E_T = \{0, 1\}$; $E_A = \{S\}$;

$$P = \{(S \rightarrow 000S111); (0S1 \rightarrow 01)\}$$

La única forma de generar sentencias es aplicando cualquier número de veces la primera regla y terminando con una aplicación de la segunda:

$$S \Rightarrow 000S111 \Rightarrow 000000S111111 \Rightarrow 0^{3n-1}0S11^{3n-1} \Rightarrow 0^{3n}1^{3n}$$

Por consiguiente, el lenguaje generado es el conjunto infinito

$$L(G) = \{0^{3n} 1^{3n} \mid n \geq 1\}$$

Si la segunda regla fuera $S \rightarrow 01$, el lenguaje sería

$$L(G) = \{0^{3n+1} 1^{3n+1} \mid n \geq 0\}$$

Como podrá comprobarse en otros ejemplos, no siempre es posible expresar de esa manera (por intensión) $L(G)$.

Ejemplo 4.2

$$E_A = \{S, A\}; E_T = \{a, b\}$$

$$P = \{(S \rightarrow abAS); (abA \rightarrow baab); S \rightarrow a; A \rightarrow b\}$$

Aquí, $L(G)$ está compuesto por cadenas que contienen (abb) y $(baab)$ intercambiándose y reproduciéndose cualquier número de veces, y terminando siempre la cadena con el símbolo a .

Ejemplo 4.3

$$E_A = \{S, A, B\}; E_T = \{a, b\}$$

$$P = \left\{ \begin{array}{ll} S \rightarrow aB & A \rightarrow bAA \\ S \rightarrow bA & B \rightarrow b \\ A \rightarrow a & A \rightarrow bS \\ A \rightarrow aS & A \rightarrow aBB \end{array} \right\}$$

El lenguaje generado es el conjunto de todas las cadenas de E_T^* que tienen igual número de a que de b , pero la demostración en este caso no es tan inmediata.

Ejemplo 4.4

$$\begin{aligned} E_T &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ E_A &= \{\langle \text{número} \rangle, \langle \text{dígito} \rangle\} \\ S &= \langle \text{número} \rangle \end{aligned}$$

$$P = \left\{ \begin{array}{lll} \langle \text{número} \rangle & \rightarrow \langle \text{dígito} \rangle \langle \text{número} \rangle & (1) \\ \langle \text{número} \rangle & \rightarrow \langle \text{dígito} \rangle & (2) \\ \langle \text{dígito} \rangle & \rightarrow 0 & (3) \\ \langle \text{dígito} \rangle & \rightarrow 1 & (4) \\ \langle \text{dígito} \rangle & \rightarrow 2 & (5) \\ \langle \text{dígito} \rangle & \rightarrow 3 & (6) \\ \langle \text{dígito} \rangle & \rightarrow 4 & (7) \\ \langle \text{dígito} \rangle & \rightarrow 5 & (8) \\ \langle \text{dígito} \rangle & \rightarrow 6 & (9) \\ \langle \text{dígito} \rangle & \rightarrow 7 & (10) \\ \langle \text{dígito} \rangle & \rightarrow 8 & (11) \\ \langle \text{dígito} \rangle & \rightarrow 9 & (12) \end{array} \right\}$$

Damos a continuación algunos ejemplos de derivaciones de sentencias, poniendo bajo el símbolo \Rightarrow el número de la regla o reglas utilizadas:

$$\begin{aligned} \langle \text{número} \rangle &\xRightarrow{2} \langle \text{dígito} \rangle \xRightarrow{3} 0 \\ \langle \text{número} \rangle &\xRightarrow{1} \langle \text{dígito} \rangle \langle \text{número} \rangle \xRightarrow{12} 9 \langle \text{número} \rangle \xRightarrow{2} 9 \langle \text{dígito} \rangle \xRightarrow{12} 99 \\ \langle \text{número} \rangle &\xRightarrow{1} \langle \text{dígito} \rangle \langle \text{número} \rangle \xRightarrow{1} \langle \text{dígito} \rangle \langle \text{dígito} \rangle \langle \text{número} \rangle \xRightarrow{1} \\ &\quad \langle \text{dígito} \rangle \langle \text{dígito} \rangle \langle \text{dígito} \rangle \langle \text{número} \rangle \xRightarrow{2} \langle \text{dígito} \rangle \langle \text{dígito} \rangle \\ &\quad \langle \text{dígito} \rangle \langle \text{dígito} \rangle \xRightarrow[4,5,6,7]{*} 1234 \end{aligned}$$

Como los símbolos terminales son los diez dígitos decimales, el lenguaje que se obtiene es el conjunto infinito de cadenas que representan en decimal a los números naturales.

Obsérvese que es la regla (1) la que permite obtener una cadena de cifras de cualquier longitud.

Ejemplo 4.5

$$E_T = \{a, b\}; E_A = \{A, S\};$$

$$P = \left\{ \begin{array}{l} S \rightarrow a S \\ S \rightarrow a A \\ A \rightarrow b A \\ A \rightarrow b \end{array} \right\} \quad \begin{array}{l} (1) \\ (2) \\ (3) \\ (4) \end{array}$$

Un análisis del tipo de sentencias que pueden derivarse nos lleva fácilmente a la conclusión de que todas terminan con el símbolo b , por aplicación de (4), y todas empiezan por a , pudiendo tener en medio cualquier número de a seguido de cualquier número de b . Obsérvese que para el lenguaje así generado puede darse una expresión regular: $L(G) = aa^*bb^*$.

5. Clasificación de las gramáticas y de los lenguajes

5.1 Gramáticas de tipo 0 ó no restringidas

La gramática definida de una manera general en el apartado 1 se llama *gramática de tipo 0 ó no restringida*. Recordemos que las reglas de escritura o producciones son de la forma $\alpha \rightarrow \beta$, con $\alpha \in E^+$ es decir, la única restricción es que no puede haber reglas de la forma $\lambda \rightarrow \beta$.

Introduciendo restricciones adicionales en las reglas se obtienen sucesivamente gramáticas cada vez más restringidas. Pasemos a definir y comentar la clasificación debida a Chomsky y aceptada universalmente.

5.2 Gramáticas de tipo 1 ó sensibles al contexto

En las gramáticas de tipo 1 las reglas son de la forma:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$$

con $A \in E_A$; $\alpha_1, \alpha_2 \in E^*$; $\beta \in E^+$

Es decir, A puede reemplazarse por β siempre que esté en el contexto de α_1 y α_2 (Obsérvese que α_1 , ó α_2 , o ambas, pueden ser la cadena vacía, λ).

Una propiedad importante de las gramáticas de tipo 1 es que *las cadenas que se van obteniendo en cualquier derivación son de longitud no decreciente*. En efecto, al definir las reglas más arriba hemos especificado que $\beta \in E^+$ es decir, $\beta \neq \lambda$ por lo que $\lg(A) = 1 \leq \lg(\beta)$, y $\lg(\alpha_1 A \alpha_2) \leq \lg(\alpha_1 \beta \alpha_2)$, o sea, que la longitud del consecuente nunca puede ser menor que la del antecedente. En el siguiente capítulo demostraremos que la inversa es también cierta, en el sentido de que, *si todas las reglas de una gramática cumplen la condición de no decrecimiento, se puede hallar una gramática equivalente con reglas sensibles al contexto*.

Salvo en el primero y el segundo, todas las gramáticas definidas en los ejemplos del apartado 4 son sensibles al contexto. En el ejemplo 4.1, es la regla $0S1 \rightarrow 01$ la que no cumple la condición, ya que sustituye S por λ en el contexto $(0, 1)$. En cuanto al segundo ejemplo, la regla $abA \rightarrow baab$ no es del

tipo sensible al contexto (lo sería si fuera $abA \rightarrow abab$); sin embargo, la longitud del antecedente es menor o igual que la del consecuente en todas las reglas, por lo que habrá una gramática equivalente sensible al contexto. En el siguiente capítulo veremos cómo se puede encontrar.

5.3 Gramáticas de tipo 2 ó libres de contexto

Las gramáticas de tipo 2 son un caso particular de las de tipo 1, con $\alpha_1 = \alpha_2 = \lambda$, es decir, las reglas son del tipo

$$A \rightarrow \beta$$

con $A \in E_A, \beta \in E^+$

Estas gramáticas, también llamadas *gramáticas de Chomsky* o *C-gramáticas* juegan un papel muy importante tanto en la lingüística como en la teoría de lenguajes de programación, por lo que ya antes de la clasificación propuesta por Chomsky fueron descubiertas por diversos autores a partir de puntos de vista bastante diferentes.

Ejemplos de gramática libres de contexto son el 4.3, el 4.4 y el 4.5.

5.4 Gramáticas de tipo 3 ó regulares

Las gramáticas de tipo 3, también llamadas *gramáticas de Kleene* o *K-gramáticas* son un caso particular de las gramáticas de tipo 2, con reglas de la forma

$$A \rightarrow aB \quad \text{o} \quad A \rightarrow a$$

con $A, B \in E_A; a \in E_T$

Un ejemplo de K-gramática es el ejemplo 4.5. Obsérvese que en ese ejemplo podíamos representar el lenguaje mediante una expresión regular; esto no es casualidad: como veremos en el capítulo 4, *los lenguajes generados por las gramáticas de tipo 3 son exactamente los lenguajes regulares que estudiábamos en el tema "Autómatas"* (y de ahí que a las K-gramáticas también se les llame gramáticas regulares).

6. Jerarquía de lenguajes

Llamaremos lenguaje de tipo 0 al generado por una gramática de tipo 0, lenguaje de tipo 1 ó sensible al contexto al generado por una gramática de tipo 1, lenguaje de tipo 2 ó libre de contexto ó C-lenguaje al generado por una gra-

mática de tipo 2, y lenguaje de tipo 3 ó regular ó K -lenguaje, o también, lenguaje de estados finitos al generado por una gramática de tipo 3.

Según se han definido las gramáticas, es evidente que toda gramática regular es libre de contexto, toda gramática libre de contexto es sensible al contexto, y toda gramática sensible al contexto es de tipo 0. Por consiguiente, si llamamos $\{L(G3)\}$, $\{L(G2)\}$, $\{L(G1)\}$ y $\{L(G0)\}$ a los conjuntos de lenguajes de cada tipo, tendremos que:

$$\{L(G3)\} \subset \{L(G2)\} \subset \{L(G1)\} \subset \{L(G0)\} \subset P(E^*)$$

7. Lenguajes con la cadena vacía

Es fácil constatar que, tal como se han definido las gramáticas, la cadena vacía, λ , no puede figurar en ningún lenguaje de tipo 1, 2 ó 3. Una gramática es, esencialmente, una expresión en un metalenguaje que permite dar una descripción finita de ciertos lenguajes (no de todos) definidos sobre E . Es obvio que si L tiene una descripción finita, $L_I = L \cup \{\lambda\}$ también puede tenerla: bastará añadir de algún modo " λ también está en L_I " a la descripción de L , y esto puede hacerse agregando $S \rightarrow \lambda$ a las reglas de la gramática que describe a L .

Ahora bien, si habíamos impuesto a las reglas de las gramáticas de tipo 1, $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, la condición de que $\beta \neq \lambda$ era para conseguir la importante propiedad de no decrecimiento. Si ahora añadimos $S \rightarrow \lambda$, será preciso que S no aparezca en el consecuente de ninguna regla si queremos que se conserve tal propiedad. A este respecto, es importante el siguiente teorema:

Teorema 7.1.

Si G es una gramática de tipo 1, 2 ó 3, puede encontrarse otra gramática equivalente, G_1 , de tipo 1, 2 ó 3 respectivamente, tal que $L(G_1) = L(G)$, y tal que su símbolo inicial, S_1 , no aparece en el consecuente de ninguna regla de G_1 . Si $G = \langle E_T, E_A, P, S \rangle$,

$$G_1 = \langle E_T, E_A \cup \{S_1\}, P_1, S_1 \rangle$$

donde

$$P_1 = P \cup \{ S_1 \rightarrow \alpha \mid (S \rightarrow \alpha) \in P \}$$

Demostración:

- a) Supongamos que $S \xrightarrow{*}_G x$, y sea $S \rightarrow \alpha$ la primera regla utilizada en esa derivación; entonces, $S \xRightarrow{G} \alpha \xrightarrow{*}_G x$. Por la definición de P_1 , $(S_1 \rightarrow \alpha) \in P_1$

de modo que $S_1 \xRightarrow{G_1} \alpha$ y como $P \subset P_1$, $\alpha \xRightarrow{G_1} x$

Por consiguiente, $S_1 \xRightarrow{G_1} x$ y $L(G) \subset L(G_1)$.

b) Supongamos que $S_1 \xRightarrow{G_1} y$ siendo $S_1 \rightarrow \beta$ la primera regla utilizada en G_1 :

$$S_1 \xRightarrow{G_1} \beta \xRightarrow{G_1} y$$

Si $S_1 \rightarrow \beta$ es una regla en G_1 , en G deberá existir la regla $S \rightarrow \beta$, por lo que $S \xRightarrow{G} \beta$.

Por otra parte, P_1 se ha definido de modo que S_1 no aparezca en el consecuente de ninguna regla, por lo que no estará incluido en α ni aparecerá en ninguna de las formas sentenciales de la derivación $\beta \xRightarrow{G_1} y$ y entonces, esta derivación será también válida en G : $\beta \xRightarrow{G} y$

Vemos así que $S \xRightarrow{G} \beta$ y por tanto, $L(G_1) \subset L(G)$.

Teniendo en cuenta el resultado anterior podemos afirmar que $L(G) = L(G_1)$.

c) Tal como se ha definido P_1 , es inmediato comprobar que si G es una gramática de tipo 1, 2 ó 3, G_1 es de tipo 1, 2 ó 3 respectivamente.

En virtud este teorema, dada una gramática cualquiera de tipo 1, 2 ó 3, G , podemos pasar a G_1 y de ésta a G_2 añadiendo la regla $S_1 \rightarrow \lambda$, con lo que tendremos $L(G_2) = L(G_1) \cup \{\lambda\}$, con G_2 del mismo tipo que G y de longitud no decreciente.

Ejemplo:

Sea G , definida por $E_A = \{S\}$; $E_T = \{0, 1\}$; $P = \{(S \rightarrow 0S1); (S \rightarrow 01)\}$.

El lenguaje es $L(G) = \{0^n 1^n \mid n \geq 1\}$.

Podemos construir G_1 , con

$$E_{A_1} = \{S, S_1\}; E_{T_1} = E_T; P_1 = \{(S_1 \rightarrow 0S1); S_1 \rightarrow 01 (S \rightarrow 0S1); (S \rightarrow 01)\}$$

siendo ahora S_1 el símbolo inicial; es fácil comprobar que $L(G_1) = L(G)$. Entonces, G_2 tendrá

$$E_{A_2} = E_{A_1} = \{S, S_1\}; E_{T_2} = E_T = \{0, 1\}; P_2 = \{(S_1 \rightarrow 0S1); (S_1 \rightarrow 01); (S \rightarrow 0S1); (S \rightarrow 01); (S_1 \rightarrow \lambda)\}$$

con lo que $L(G_2) = L(G) \cup \{\lambda\} = \{0^n 1^n \mid n \geq 0\}$.

De una manera general, del Teorema 7.1 se obtiene el siguiente
Corolario.

Si L es un lenguaje de tipo 1, 2 ó 3, entonces $L \cup \{\lambda\}$ y $L - \{\lambda\}$ son lenguajes de tipo 1, 2 ó 3 respectivamente.

8. Resumen

En este capítulo hemos definido los conceptos de gramática, lenguaje generado por una gramática y gramáticas equivalentes. Hemos visto la clasificación de las gramáticas según las restricciones impuestas a sus producciones y cómo esta clasificación da lugar a una jerarquía de lenguajes. Finalmente, hemos estudiado la manera de modificar una gramática para que el lenguaje contenga la cadena vacía sin cambiar de tipo.

9. Notas histórica y bibliográfica

La teoría de lenguajes tiene su origen en un campo inicialmente bastante alejado de la informática: la lingüística. Los lingüistas distinguen, tradicionalmente, entre *gramática particular* (propiedades de lenguajes concretos, como frecuencia de vocablos, reglas sintácticas, etc.) y *gramática universal* (propiedades generales que puedan aplicarse a cualquier lenguaje humano) (Chomsky, 1967).

Los lingüistas de la llamada "escuela estructuralista americana" habían elaborado por los años 50 algunas ideas informales acerca de la gramática universal. Por ejemplo, si un lenguaje (natural) es un conjunto innumerable de frases, para describirlo debería establecerse una *gramática generativa* o conjunto de reglas que subyacen en la composición de frases correctas y una *descripción estructural* para cada frase que permitiese explicar cómo puede componerse tal frase a partir de la gramática. El primer trabajo sobre la formalización de estos conceptos fue obra de Chomsky (1956), quien sin duda es la figura más destacada de la lingüística moderna, tanto por desarrollar sus fundamentos matemáticos (Chomsky y Miller, 1958; Chomsky, 1959) como por sus teorías sobre el origen y la naturaleza de los lenguajes naturales (Chomsky, 1968, 1975), aunque estas son más discutidas. (El lector puede encontrar, traducidos al español, dos libros del más conocido crítico de Chomsky: Luria (1974a, b).) Por ejemplo, las gramáticas generativas formalizadas permiten explicar el "carácter creativo" de los lenguajes naturales, es decir, el hecho de que dispongan de mecanismos recursivos que les permiten expresar un número potencialmente infinito de ideas, sentimientos, etc. La

falta de un formalismo para estudiar estos mecanismos había inclinado previamente a ciertos lingüistas de la escuela conductista a negar tal propiedad, y a otros, como Saussure, a considerarla como algo ajeno al campo de la lingüística (Chomsky, 1967).

En el campo de la informática, poco después de la aparición de las primeras publicaciones de Chomsky, el concepto de gramática formal adquirió gran importancia para la especificación de los lenguajes de programación; concretamente, se definió formalmente la sintaxis del lenguaje ALGOL 60 (con ligeras modificaciones sobre su versión primitiva) mediante una gramática libre de contexto (Naur, 1963). Ello condujo rápidamente, de una manera natural, al diseño riguroso de algoritmos de compilación (Randell y Russell, 1964).

Desde una perspectiva bastante ambiciosa, hace tiempo se piensa que, puesto que todos los lenguajes de programación son lenguajes artificiales susceptibles de formalización, la teoría de lenguajes podría ser la base de una "teoría general de la programación" (Harrison, 1965) o de una "ciencia de la programación" (Gries, 1981).

Finalmente, y enlazando con el campo de la lingüística, la teoría de lenguajes es de gran utilidad para el trabajo en un campo de la inteligencia artificial: el procesamiento de lenguajes naturales (comprensión, generación y traducción) (Hirst, 1981; Tennant, 1981; Carbonell *et al.*, 1981).

En cuanto a orientaciones bibliográficas para estudiar con mayor profundidad la teoría de lenguajes formales, tres libros clásicos y recomendables son el de Hopcroft y Ullman (1969), el de Harrison (1978) y el de Gross y Lentin (1967). El ejemplo 4.2 (como también los ejemplos 4.3 y 5.4 del capítulo siguiente, relacionados con él) está tomado del primero, donde pueden encontrarse algunos detalles y demostraciones que aquí omitimos. En otra obra posterior de Hopcroft y Ullman (1979) se incluyen también aspectos de la teoría de la computabilidad, y en la misma línea se orienta el libro, más completo, de Denning *et al.* (1978).

Para referencias más modernas y más ligadas a la aplicación de la teoría a los lenguajes de programación, véase el apartado 7 del capítulo 6.

10. Ejercicios

10.1 Muchas veces interesa que los árboles de derivación en una gramática sean binarios, es decir, que de cada nodo sólo puedan salir uno o dos arcos. ¿Cómo deberán ser las reglas de escritura para que esto sea posible? Modificar la gramática de los ejemplos del capítulo 1 para que sus árboles sean binarios.

10.2 Dada la gramática $E_A = \{S, A\}$; $E_T = \{0, 1\}$;

$$P = \{(S \rightarrow 0A); (A \rightarrow 0A); (A \rightarrow 1S); (A \rightarrow 0)\},$$

- a) ¿de qué tipo es?;
- b) expresar de algún modo el lenguaje que genera;
- c) hallar otra gramática que genere el mismo lenguaje más la cadena vacía.

10.3 Definir una gramática que permita generar todos los números racionales escritos en decimal con el formato: $\langle \text{signo} \rangle \langle \text{parte entera} \rangle \langle \text{parte fraccionaria} \rangle$.

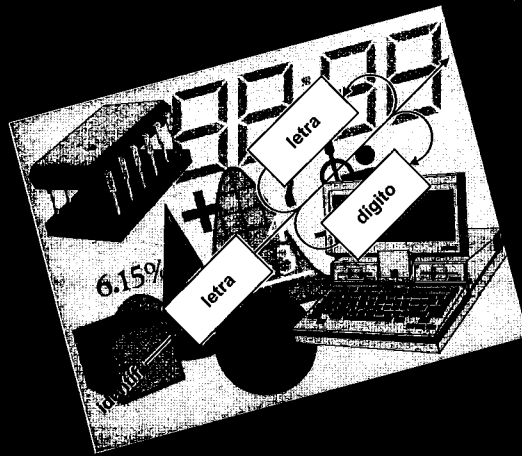
10.4 Definir una gramática que permita generar "identificadores": secuencias de letras y dígitos que empiezan siempre por una letra.

10.5 Definir gramáticas para describir los lenguajes de la lógica de proposiciones y de la lógica de predicados.

10.6 Modificar las gramáticas de los ejemplos del apartado 4 para que se obtengan los mismos lenguajes con la cadena vacía.

Fundamentos de informática

3



De algunas propiedades de los lenguajes formales

1. Introducción

En el estudio de los lenguajes se plantean problemas que tienen o no solución dependiendo del tipo de gramática. Por ejemplo, para una gramática, G , de reglas sensibles al contexto, el problema de saber si $L(G)$ es vacío, finito o infinito es, en general, indecidible, mientras que para una gramática libre de contexto es decidible, es decir, existe un algoritmo que, aplicado a G , produce una de las tres repuestas; naturalmente, este problema será indecidible para las gramáticas de tipo 0 y decidible para las regulares. Otro ejemplo es el de averiguar si dos gramáticas son equivalentes, problema sólo decidible para las gramáticas regulares.

Al aumentar el grado de generalidad de la gramática, es decir, al ir desde el tipo 3 hacia el tipo 0, el estudio se hace más complejo y aumenta el número de problemas indecidibles.

Desde el punto de vista de aplicación práctica tanto a los lenguajes naturales como a los de programación, son particularmente importantes las gramáticas libres de contexto. Este capítulo se dedica, esencialmente, a dos características asociadas a tales gramáticas: la *recursividad* y la *ambigüedad*.

La recursividad, como veremos, es una propiedad válida no sólo para las gramáticas libres de contexto, sino también para las sensibles al contexto, ya que es una consecuencia del no decrecimiento de las cadenas. La ambigüedad sólo puede definirse y estudiarse cómodamente para las gramáticas de tipo 2 (y 3), en las que pueden describirse las derivaciones mediante árboles.

En aras de la brevedad omitiremos las demostraciones de algunos teoremas, que pueden encontrarse en la bibliografía referenciada en el apartado 9.

2. No decrecimiento en las gramáticas sensibles al contexto

En el capítulo anterior vimos que como consecuencia de la misma definición de reglas sensibles al contexto ($\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, $\beta \neq \lambda$) se desprende inmediatamente que la longitud de las cadenas derivadas por aplicación de tales reglas no puede disminuir. Nos proponemos ahora demostrar la inversa, que ya no es tan evidente:

Teorema 2.1.

Dada una gramática G_1 con reglas de longitud no decreciente, puede encontrarse otra gramática G_2 equivalente a G_1 cuyas reglas son sensibles al contexto.

Demostración

Sea $\alpha \rightarrow \beta$ una regla de G_1 , y sea

$$\alpha = p_1 p_2 \dots p_\ell \text{ y } \beta = q_1 q_2 \dots q_{\ell+m}$$

con $p_i q_j \in E$ y $m \geq 0$.

Sustituiremos esta regla por un conjunto de reglas sensibles al contexto tales que juntas producen la derivación $\alpha \xrightarrow[G_2]{*} \beta$ sin aumentar la capacidad generativa.

Para ello ampliamos el alfabeto auxiliar de G_1 con un símbolo R y le añadimos también un símbolo P_i por cada $p_i \in E_T$

Sustituimos en principio

$$p_1 p_2 p_3 \dots p_\ell \rightarrow q_1 q_2 q_3 \dots q_{\ell+m}$$

[illegible]

a) Con este conjunto de reglas se obtiene la derivación

b) Al ser R un símbolo auxiliar no pueden producirse más sentencias que las que pueda producir $\alpha \rightarrow \beta$.

Ejemplo:

$$\begin{aligned} abA &\rightarrow Rba \\ RbA &\rightarrow RaA \\ RaA &\rightarrow Raab \\ Raab &\rightarrow baab \end{aligned}$$
$$E_T = \{a, b\}; E_A = \{S, A, R, P_1, P_2\};$$

$$P = \left\{ \begin{array}{ll} S & \rightarrow P_1 P_2 A S \\ P_1 P_2 A & \rightarrow R P_2 A \\ R P_2 A & \rightarrow R P_1 A \\ R P_1 A & \rightarrow R P_1 P_1 P_2 \\ R P_1 P_1 P_2 & \rightarrow P_2 P_1 P_1 P_2 \end{array} \quad \begin{array}{l} S \rightarrow P_1 \\ A \rightarrow P_2 \\ P_1 \rightarrow a \\ P_2 \rightarrow b \end{array} \right\}$$

3. Recursividad de los lenguajes sensibles al contexto

Una gramática es un algoritmo para generar un lenguaje, pero un problema de gran importancia es el del reconocimiento, es decir, dados un lenguaje $L \subset E^*$ y una cadena $x \in E^*$, ¿existe un algoritmo para determinar si $x \in L$ ó $x \notin L$? Si, por ejemplo, L es un lenguaje de programación y x un programa, es importante poder determinar si el programa es sintácticamente correcto ($x \in L$) o no ($x \notin L$). El problema del reconocimiento está íntimamente ligado al concepto de recursividad (ver tema "Algoritmos", capítulo 6). En efecto, decir que un lenguaje $L \subset E^*$ es recursivo es lo mismo que decir que existe un algoritmo para calcular la función característica de todo $x \in E^*$, o, lo que es lo mismo, para determinar si $x \in L$ o $x \notin L$.

Teorema 3.1.

Todo lenguaje generado por una gramática de reglas sensibles al contexto es recursivo.

Aunque no daremos una demostración de este importante teorema, puede fácilmente intuirse que es una consecuencia de la propiedad de no decrecimiento. En efecto, un lenguaje de tipo 0 es recursivamente numerable, ya que existe un algoritmo (la gramática de tipo 0) para generar sus elementos (sentencias). Dado un elemento $x \in L$, podemos compararlo con cada uno de los elementos generados hasta comprobar que coinciden, pero si $x \notin L$ habría que generar las infinitas sentencias para ver que efectivamente $x \notin L$; es decir, en general, un lenguaje de tipo 0 no es recursivo. Sin embargo, si el lenguaje cumple la propiedad de no decrecimiento (tipo 1 en adelante), tenemos un algoritmo para generar primero todas las sentencias de longitud 1, luego las de longitud 2, etc., si $lg(x) = \ell$, generaremos todas las sentencias de longitud ℓ , y si x no se encuentra entre ellas, entonces $x \notin L$.

Podemos preguntarnos si la inversa es también cierta, es decir, si todo lenguaje recursivo puede ser generado por una gramática sensible al contexto. La contestación es "no":

Teorema 3.2.

Existen lenguajes recursivos que no son sensibles al contexto.

4. Árboles de derivación para las gramáticas libres de contexto

En el capítulo 1 ya hemos utilizado árboles como un método gráfico para ilustrar las derivaciones en ciertas gramáticas.

Definición 4.1.

Un *árbol* es un conjunto finito de *nodos* unidos por *arcos* orientados (diremos que un arco *sale* del nodo n_i y *entra* en el nodo n_j), cumpliéndose tres condiciones:

1. Existe un nodo, y sólo uno, llamado *raíz*, en el que no entra ningún arco.
2. Para todo nodo n_m existe una secuencia de arcos, y sólo una, tal que el primero sale de la raíz y entra en n_i , el segundo sale de n_i y entra en n_{i+1} , etc., y el último entra en n_m .
3. En cada nodo (salvo la raíz) entra un arco y sólo uno.

Un nodo n_j es *descendiente directo* (o "hijo") de otro nodo n_i si existe un arco que sale de n_i y entra en n_j . Un nodo n_m es *descendiente* de otro n_o si existe una secuencia n_1, n_2, \dots, n_{m-1} tal que n_m es descendiente directo de n_{m-1} , n_{m-1} lo es de n_{m-2} , ..., n_1 lo es de n_o .

Se llaman *hojas* a los nodos que no tienen ningún descendiente.

Entre los nodos de un árbol se puede establecer una *relación de orden*: todos los descendientes directos de n se pueden ordenar de *izquierda a derecha*, y si n_1 está a la izquierda de n_2 todos los descendientes de n_1 estarán a la izquierda de n_2 .

Definición 4.2.

Dada una gramática libre de contexto $G = \langle E_A, E_T, P, S \rangle$ un árbol es un *árbol de derivación* o *árbol sintáctico* en G si:

1. Cada nodo tiene una etiqueta que es un símbolo de $E = E_A \cup E_T$:
 $\text{Eti}(n) \in E$
2. $\text{Eti}(\text{raíz}) = S$
3. Si n no es una hoja, $\text{Eti}(n) \in E_A$.
4. Si n_1, n_2, \dots, n_k son todos los descendientes directos de n de izquierda a derecha y $\text{Eti}(n) = A$, $\text{Eti}(n_1) = \ell_1$, $\text{Eti}(n_2) = \ell_2$, ..., $\text{Eti}(n_k) = \ell_k$, entonces $(A \rightarrow \ell_1 \ell_2, \dots, \ell_k) \in P$.

Llamaremos resultado de un árbol de derivación a la cadena compuesta por las etiquetas de las hojas leídas de izquierda a derecha.

Ejemplo 4.3

$$E_A = \{S, A, B\}; E_T = \{a, b\}$$

$$P = \left\{ \begin{array}{lll} S \rightarrow aB & (1) & A \rightarrow a \quad (5) \\ S \rightarrow bA & (2) & A \rightarrow bAA \quad (6) \\ S \rightarrow aBS & (3) & B \rightarrow b \quad (7) \\ S \rightarrow bAS & (4) & B \rightarrow aBB \quad (8) \end{array} \right\}$$

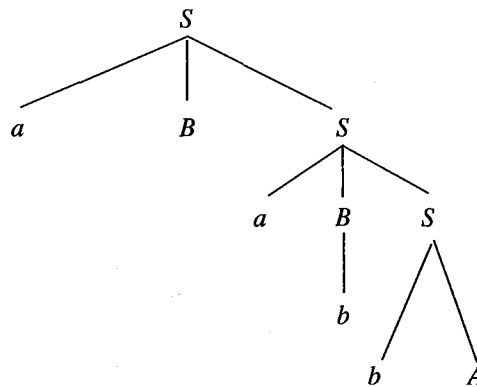


Figura 3.1.

El árbol de la figura 3.1 tiene como resultado la cadena *aBabbA*, y corresponde a la derivación

$$S \xRightarrow{3} aBS \xRightarrow{3} aBaBS \xRightarrow{7} aBabS \xRightarrow{2} aBabbA$$

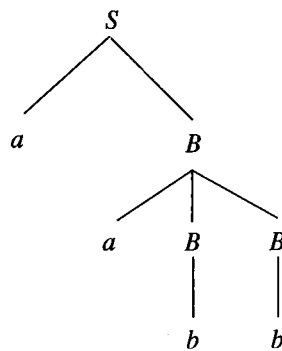


Figura 3.2.

El árbol de la figura 3.2 tiene como resultado la sentencia *aabb*, y corresponde a la derivación

$$S \xRightarrow{1} aB \xRightarrow{8} aaBB \xRightarrow{7} aabB \xRightarrow{7} aabb$$

Teorema 4.4.

Dada una gramática libre de contexto, G , $S \xRightarrow{*}_G \alpha$ si y sólo si existe un árbol de derivación en G con resultado α .

5. Ambigüedad en las gramáticas libres de contexto

Definición 5.1.

Una gramática libre de contexto, G , se dice que es ambigua si existe al menos una sentencia en $L(G)$ que puede obtenerse por dos o más derivaciones distintas, o lo que es lo mismo, le corresponden dos o más árboles diferentes.

Ejemplo 5.2

Los árboles de derivación suelen utilizarse en lingüística para el análisis sintáctico de oraciones. A la oración (ambigua) "Juan ve a Luis con gafas" puede corresponderle el árbol de la figura 3.3a o el de la figura 3.3b, según se interprete que es Juan o Luis, respectivamente, quien usa gafas.

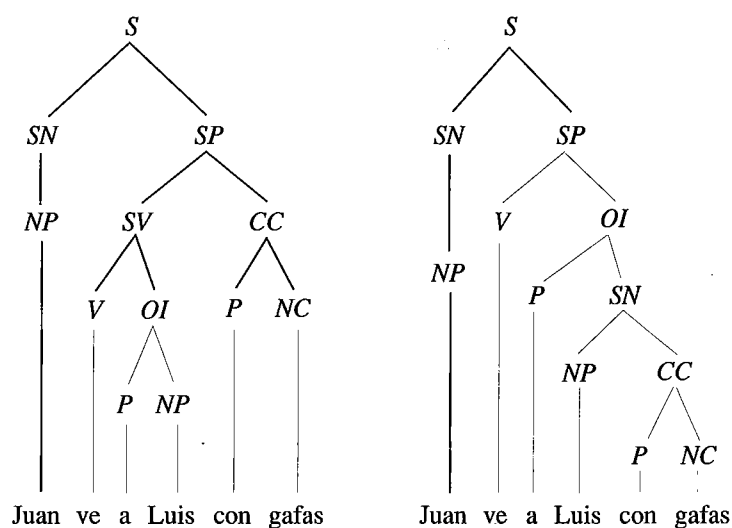


Figura 3.3.

Ejemplo 5.3

Consideremos la gramática definida por:

$$E_A = \{\langle EA \rangle\}; E_T = \{ |, +, * \}; S = \langle EA \rangle$$

("EA" significa "expresión aritmética")

$$P = \left\{ \begin{array}{l} \langle EA \rangle \rightarrow \langle EA \rangle + \langle EA \rangle \\ \langle EA \rangle \rightarrow \langle EA \rangle * \langle EA \rangle \\ \langle EA \rangle \rightarrow | \\ \langle EA \rangle \rightarrow \langle EA \rangle \rightarrow | \end{array} \right\}$$

Si admitimos que una sucesión de n "|" representa el número natural n , esta gramática genera sentencias que representan combinaciones de los números naturales con las operaciones de suma y producto. Consideremos la sentencia $| + | * | |$; la derivación puede hacerse mediante el árbol de la figura 3.4.

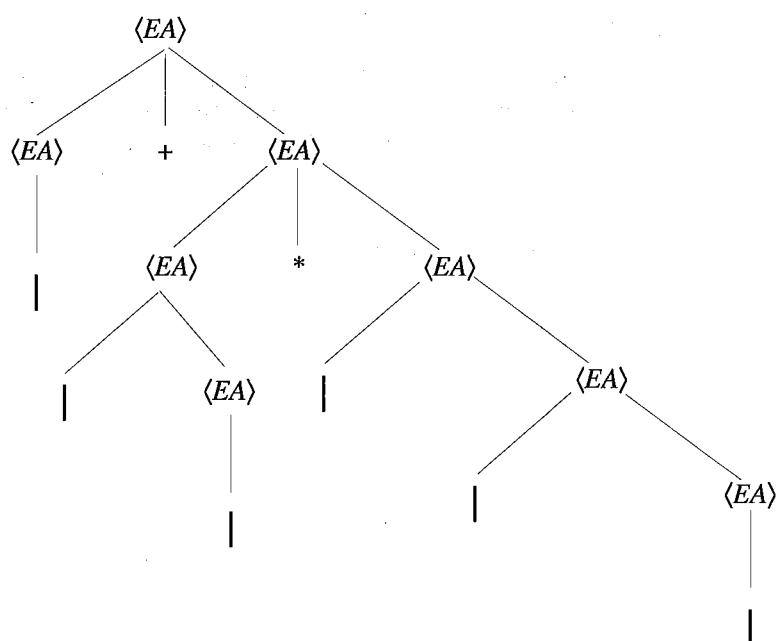


Figura 3.4.

O también mediante el de la figura 3.5.

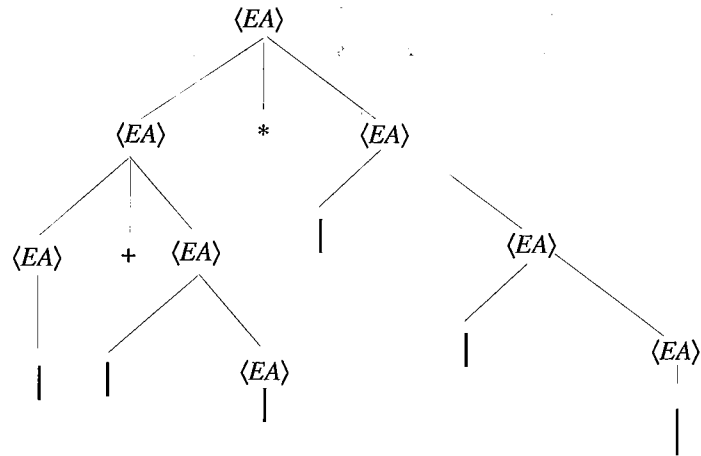


Figura 3.5.

El primero corresponde a la interpretación $|+|(||*|)|$ (prioridad de *), es decir, 7, mientras que el segundo corresponde a $(|+|)|*|)|$ (prioridad de +), es decir, 9.

Puede ocurrir que un mismo lenguaje puede ser generado por una gramática ambigua y por otra gramática no ambigua. Un lenguaje es inherentemente ambiguo si todas las gramáticas que lo generan son ambiguas.

Ejemplo 5.4

La gramática del ejemplo 4.3 del capítulo 2 es ambigua. En efecto, la sentencia *aabbab*, por ejemplo, puede derivarse según indican los dos árboles de la figura 3.6.

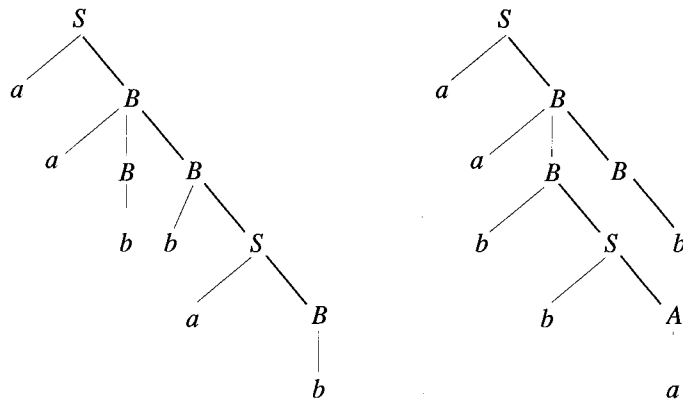


Figura 3.6.

Sin embargo, la gramática del ejemplo 4.3 de este capítulo es equivalente a ella y no ambigua, por lo que el lenguaje generado (conjunto de cadenas que tienen igual número de a 's que de b 's) no es inherentemente ambiguo. La derivación de la misma cadena en esta segunda gramática se representa por el árbol de la figura 3.7.

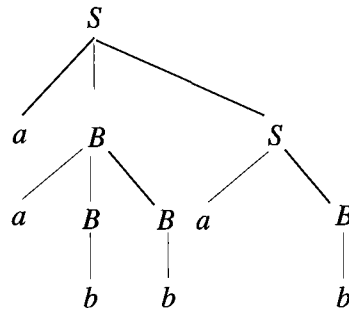


Figura 3.7.

Ejemplo 5.5

Una ambigüedad similar a la del ejemplo 5.3 puede ocurrir en las expresiones aritméticas decimales. Con una gramática definida por

$$\begin{aligned}
 E_A &= \{ \langle E \rangle, \langle CTE \rangle, \langle DIG \rangle \} \\
 E_T &= \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, +, * \} \\
 S &= \{ \langle EA \rangle \}
 \end{aligned}$$

y las reglas

$$\begin{aligned}
 \langle EA \rangle &\rightarrow \langle EA \rangle + \langle EA \rangle \\
 \langle EA \rangle &\rightarrow \langle EA \rangle * \langle EA \rangle \\
 \langle EA \rangle &\rightarrow \langle CTE \rangle \\
 \langle CTE \rangle &\rightarrow \langle CTE \rangle \langle DIG \rangle \\
 \langle CTE \rangle &\rightarrow \langle DIG \rangle \\
 \langle DIG \rangle &\rightarrow 0 \\
 \langle DIG \rangle &\rightarrow 1 \\
 \langle DIG \rangle &\rightarrow 2 \\
 \langle DIG \rangle &\rightarrow 3 \\
 \langle DIG \rangle &\rightarrow 4 \\
 \langle DIG \rangle &\rightarrow 5 \\
 \langle DIG \rangle &\rightarrow 6 \\
 \langle DIG \rangle &\rightarrow 7 \\
 \langle DIG \rangle &\rightarrow 8 \\
 \langle DIG \rangle &\rightarrow 9
 \end{aligned}$$

el lector puede fácilmente comprobar (de igual manera que en el ejemplo 5.3) la ambigüedad de, por ejemplo, " $1 + 2 * 3$ ".

Sabemos que una notación eficaz para eliminar la ambigüedad en casos como éste es el uso de los paréntesis. Podemos incluir esta posibilidad en nuestra gramática ampliando E_T con "(" y ")" y P con la producción " $\langle EA \rangle \rightarrow (\langle EA \rangle)$ ". Ahora, existe un árbol único para " $1 + (2*3)$ " y otro para " $(1 + 2)*3$ ". Pero ello no impide que la sentencia " $1 + 2*3$ " siga siendo válida en nuestro lenguaje y ambigua.

Sabemos también que la ambigüedad en una sentencia como " $1 + 2*3$ " se elimina estableciendo una prioridad entre los operadores. Concretamente, se suele dar prioridad a "*" sobre "+", de manera que la sentencia se interpreta como " $1 + (2*3)$ ". Pues bien, podemos establecer una gramática equivalente a la anterior que refleja esta prioridad y que, por tanto, no es ambigua. Para ello, ampliamos E_A con dos nuevos símbolos: " $\langle TERM \rangle$ " (término) y " $\langle FACT \rangle$ " (factor), y definimos las nuevas producciones así:

$$\begin{array}{ll} \langle EA \rangle & \rightarrow \langle EA \rangle + \langle TERM \rangle \\ \langle EA \rangle & \rightarrow \langle TERM \rangle \\ \langle TERM \rangle & \rightarrow \langle TERM \rangle * \langle FACT \rangle \\ \langle TERM \rangle & \rightarrow \langle FACT \rangle \\ \langle FACT \rangle & \rightarrow \langle CTE \rangle \\ \langle FACT \rangle & \rightarrow (\langle EA \rangle) \\ \langle CTE \rangle & \rightarrow \langle CTE \rangle \langle DIG \rangle \\ \langle CTE \rangle & \rightarrow \langle DIG \rangle \\ \langle DIG \rangle & \rightarrow 0 \\ \dots\dots\dots & \\ \langle DIG \rangle & \rightarrow 9 \end{array}$$

Ahora, la sentencia $1 + 2 * 3$ sólo tiene un árbol, el de la figura 3.8. Y, en general, "*" tiene siempre precedencia sobre "+", excepto cuando se usan paréntesis para invalidar tal precedencia.

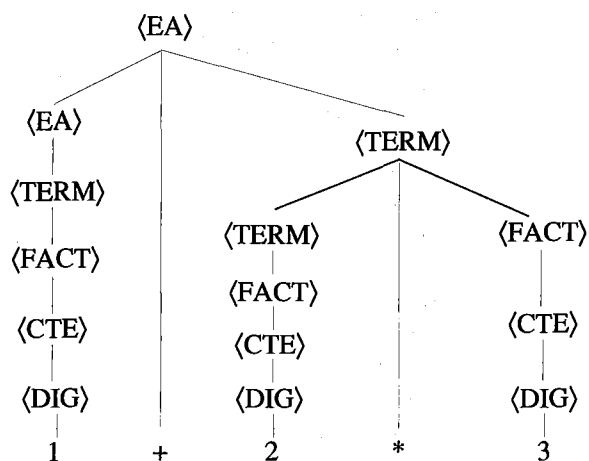


Figura 3.8.

El problema de la ambigüedad, en general, es indecidible;

Teorema 5.6.

No existe un algoritmo que, aplicado a una C -gramática arbitraria, permita decidir si la gramática es ambigua o no.

6. Resumen

Los lenguajes de tipo 0 son recursivamente numerables pero no todos son recursivos. Los lenguajes de tipo 1 (y 2 y 3) son recursivos: existe un algoritmo para decidir si $x \in L$ ó $x \notin L$. Sin embargo, hay lenguajes recursivos que no son de tipo 1. Es decir:

$$\{L(G3)\} \subset \{L(G2)\} \subset \{L(G1)\} \subset \{L_{\text{recursivos}}\} \subset \{L(G0)\} = \{L_{\text{recursivamente num}}\}$$

En los lenguajes libres de contexto las derivaciones se pueden representar gráficamente mediante árboles de derivación. A cada derivación corresponde un árbol y viceversa. Si alguna sentencia puede derivarse de dos o más formas diferentes se dice que la gramática es ambigua.

El problema de la ambigüedad es, en general, indecidible. No obstante, en muchos casos es posible, dada una gramática ambigua, encontrar otra equivalente a ella y no ambigua.

7. Notas histórica y bibliográfica

La mayor parte de los resultados concernientes a los lenguajes sensibles al contexto y, especialmente, a los libres de contexto, así como sus aplicaciones tanto a lenguajes naturales como a lenguajes de programación aparecieron durante los años 60. Las primeras propiedades indecidibles de las C -gramáticas se publicaron en Bar-Hillel et al. (1961), y la indecidibilidad de la ambigüedad se demostró por Cantor (1962), Floyd (1962) y Chomsky y Schutzenberger (1963), independientemente. Floyd (1964) publicó una revisión sobre la aplicación de las gramáticas formales a los lenguajes de programación.

La bibliografía, especialmente sobre C -gramáticas, es muy amplia; como botón de muestra podemos citar el libro de Ginsburg (1966), del cual se puede encontrar un resumen en Ginsburg (1968).

Hay gran cantidad de resultados sobre propiedades indecidibles de las C -gramáticas y sobre simplificación de las mismas o reducción a unas llamadas "formas normales", que pueden estudiarse en libros específicos, como el citado más arriba, o de carácter general, como los cinco recomendados en el capítulo anterior.

8. Ejercicios

8.1 Dada la gramática definida por

$$E_A = \{S\}; E_T = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$P = \left\{ \begin{array}{l} S \rightarrow 12S \\ 12 \rightarrow 345 \\ 45 \rightarrow 678 \\ S \rightarrow 9 \end{array} \right\}$$

construir una gramática equivalente con reglas sensibles al contexto.

8.2 Dada la gramática definida por

$$E_A = \{S, A, B\}; E_T = \{0, 1\}$$

$$P = \left\{ \begin{array}{ll} S \rightarrow 0A & B \rightarrow 1S0 \\ S \rightarrow 1B & B \rightarrow 0 \\ A \rightarrow 0S & \end{array} \right\}$$

determinar, de las siguientes sentencias, cuáles pertenecen y cuáles no al lenguaje: 00, 10; 100; 1100; 1010.

8.3 ¿De qué tipo es una gramática cuyas reglas son todas de la forma $A \rightarrow B$ ó $A \rightarrow x$, con $A, B \in E_A$, $x \in E_T^*$?; demostrar que siempre puede encontrarse una gramática regular equivalente.

8.4 Con $E_A = \{S\}$; $E_T = \{a, b, c\}$, la gramática G_1 con reglas

$$P_1 = \left\{ \begin{array}{l} S \rightarrow aSa \\ S \rightarrow bSb \\ S \rightarrow aaSaa \\ S \rightarrow c \end{array} \right\}$$

es una gramática ambigua, mientras que G_2 , con

$$P_2 = \left\{ \begin{array}{l} S \rightarrow aSa \\ S \rightarrow bSb \\ S \rightarrow c \end{array} \right\}$$

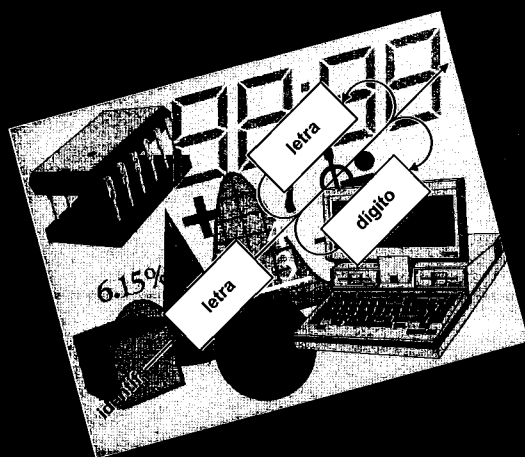
es equivalente a G_1 y no es ambigua. Comprobarlo haciendo algunas derivaciones. Describir informalmente la forma general de las sentencias del lenguaje.

8.5 Definir una gramática para las expresiones aritméticas como las del ejemplo 5.5, pero tal que "+" tenga preferencia sobre "*".

8.6 Definir una gramática para expresiones algebraicas que incluya tanto "constantes" como "identificadores" (ver el ejercicio 10.4 del capítulo anterior).

Fundamentos de informática

4



Lenguajes y Autómatas

1. Introducción

En este capítulo vamos a plantear el problema del reconocimiento de lenguajes desde un punto de vista material, ya que estudiaremos las relaciones entre el tipo de lenguaje y la estructura de la máquina capaz de reconocerlo; estas relaciones pueden enfocarse en dos sentidos:

- a) dada una gramática, G , ¿qué estructura deberá tener una máquina, M , tal que $L(M) = L(G)$? (es decir, lenguaje reconocido por M igual a lenguaje generado por G);
- b) dada una máquina, M . ¿Cuál será la gramática, G , tal que $L(G) = L(M)$?

De anteriores temas conocemos dos estructuras de máquina que pueden ser utilizadas como reconocedores de cadenas de símbolos: el autómata finito (tema "Autómatas") y la máquina de Turing (tema "Algoritmos"). Veremos aquí que la clase de lenguajes que pueden ser reconocidos por autómatas fini-

tos es precisamente la clase de lenguajes que pueden ser generados por una gramática de tipo 3 (regular), y análogamente para las MT y las gramáticas de tipo 0. Para los dos tipos intermedios de gramática definiremos unos autómatas que pueden considerarse como restricciones de la máquina de Turing o como extensiones del autómata finito. Tendremos así una jerarquía de máquinas paralela a la jerarquía de lenguajes¹.

Reflexione el lector en el hecho de que las "clases de lenguajes" (la clase de los lenguajes generados por una gramática regular, la clase de los lenguajes reconocidos por un autómata finito, etc.) son subconjuntos de $P(E^*)$ (conjunto de las partes de E^*).

El desarrollo completo de las ideas expuestas exige la demostración de una serie de teoremas acompañada de las correspondientes construcciones (dada una gramática de tipo 0, diseñar la correspondiente MT y viceversa, etc.) que alargaría excesivamente este tema, por lo que nos contentaremos con desarrollar únicamente el caso más sencillo (gramáticas regulares y autómatas finitos), limitándonos a enunciar los teoremas en los otros tres.

2. Lenguajes de tipo 0 y máquinas de Turing

2.1 Reconocedor de Turing

En el capítulo 5, apartado 2, del tema "Algoritmos", se define la parte de control de una MT como

$$T - Q = \langle E, (E \times M) \cup (\text{Stop}), Q, f, g \rangle$$

Al igual que en el tema "Autómatas" (capítulo 4) adaptábamos la definición general de autómata finito para especializarlo como reconocedor, adaptaremos también la definición de $T - Q$. Primero, explicaremos de una manera informal lo que entendemos por máquina de Turing reconocedora o reconocedor de Turing.

Un *reconocedor de Turing* será una MT que, inicializada en un estado inicial predeterminado, q_1 , y teniendo la cadena $x \in E^*$ en su cinta (descripción instantánea inicial $0q_1x0$) opera de tal modo que después de un tiempo finito se para, tras escribir un 1, si $x \in L$. Si $x \notin L$ pueden ocurrir dos cosas: escribe 0 y se para (rechaza x), o no se para. (Suponemos que $\{0, 1\} \in E$).

$T - Q$ es un autómata finito que siempre se podrá describir como una máquina de Moore, con una función de salida:

$$h: Q \rightarrow (E \times M) \cup (\text{Stop})$$

¹ Obsérvese que un AF se puede considerar como un caso muy particular de MT que sólo puede leer de la cinta, ésta se desplaza en un sólo sentido, y no para.

Definiremos los estados finales de aceptación como aquellos para los que la función h es "stop", siendo la h del estado anterior "1". Tendremos así un conjunto $F_A \subset Q$ de estados de aceptación:

$$F_A = \{q_A \mid h(q_A) = \text{Stop y } h(q_A) = 1 \\ \text{siendo } q_A \text{ tal que } f(e, q_A) = q_A \text{ para algún } e \notin E\}$$

El control de un reconocedor de Turing será definido como

$$R_{T-Q} = \langle E, Q, f, q_1, F_A \rangle$$

y el reconocedor será $R_{MT} = R_{T-Q} + \text{cinta}$.

2.2 Lenguaje aceptado por un reconocedor de Turing

Podemos ya definir el lenguaje aceptado por un R_{MT} :

$$L(R_{MT}) = \{x \in E^* \mid \text{Res } R_{MT} (0q_1x0) = 0yq_Az0; q_A \in F_A; y, z \in E\}$$

y enunciar los dos teoremas que establecen la *equivalencia entre la clase de lenguajes aceptados por algún R_{MT} y la clase de lenguajes generados por una gramática de tipo 0*.

2.3 Teorema MT1

Para toda gramática de tipo 0, G_0 , existe un reconocedor de Turing, R_{MT} , tal que $L(R_{MT}) = L(G_0)$

Obsérvese que, en general, no tiene por qué existir R_{MT} tal que $L(R_{MT}) = E^* - L(G_0)$; esto sólo ocurrirá en el caso de que $L(G_0)$ sea recursivo, y en tal caso R_{MT} puede ser el mismo R_{MT} , definiendo los estados de aceptación por $h(q_A) = 0$

2.4 Teorema MT2

Para todo reconocedor de Turing, R_{MT} , existe una gramática de tipo 0, G_0 , tal que $L(G_0) = L(R_{MT})$.

2.5 Conclusión

El teorema MT1 viene a decir que $\{L(G_0)\} \subset \{L(R_{MT})\}$, y el MT2, que $\{L(R_{MT})\} \subset \{L(G_0)\}$. Además, sabemos que la clase de lenguajes reconocibles por un R_{MT} coincide con la clase de los lenguajes recursivamente nume-

rables (esto es una consecuencia de la hipótesis de Turing) y que los lenguajes recursivos son un subconjunto de éstos, luego, como conclusión, escribiremos:

$$\{ L_{\text{recurs.}} \} \subset \{ L_{\text{recurs. num.}} \} = \{ L(R_{\text{MT}}) \} = \{ L(G_0) \}$$

3. Lenguajes sensibles al contexto y autómatas limitados linealmente

3.1 Autómata limitado linealmente

Un *autómata limitado linealmente*, ALL , es una MT cuya cabeza no puede desplazarse fuera de los límites entre los que se sitúa inicialmente la cadena de entrada. Para señalar tales límites, se incluye en E un símbolo especial como marcador, $\#$ (figura 4.1)

Un reconocedor limitado linealmente, R_{ALL} , se definirá por estados finales de aceptación con un estado inicial q_1 , igual que un R_{MT} .

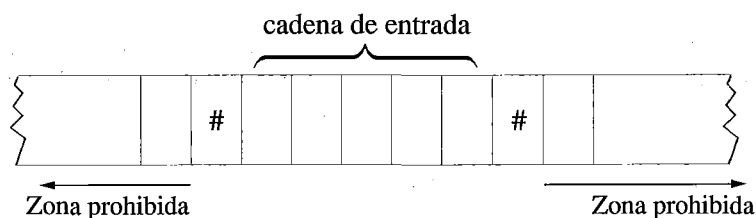


Figura 4.1.

3.2 Lenguaje aceptado por un reconocedor limitado linealmente

La definición del *lenguaje aceptado por un R_{ALL}* es análoga a la de $L(R_{MT})$, salvo que $\#$ no puede utilizarse como símbolo en las cadenas de entrada y que se preserva la distancia entre los dos marcadores:

$$L(R_{ALL}) = \{ x \in (E - \#)^* \mid \text{Res } R_{ALL}(\#q_1x\#) = \# \text{ y } q_A z \# \\ q_A \in F_A; y, z \in (E - \#)^*; \log(x) = \lg(yz) \}$$

3.3 Teorema ALL1

Para toda gramática sensible al contexto, G_1 , existe un reconocedor limitado linealmente, R_{ALL} , tal que $L(R_{ALL}) = L(G_1)$.

Como las gramáticas de tipo 1 generan lenguajes recursivos, siempre existirá también R'_{ALL} tal que $L(R'_{ALL}) = E^* - L(G1)$.

3.4 Teorema ALL2

Para todo reconocedor limitado linealmente, R_{ALL} , existe una gramática sensible al contexto, $G1$, tal que $L(G1) = L(R_{ALL})$.

3.5 Conclusión

De los teoremas anteriores deducimos:

$$\{ L(R_{ALL}) \} = \{ L(G1) \}$$

4. Lenguajes libres de contexto y autómatas de pila

4.1 Autómata de pila

Una pila es un tipo de almacenamiento en que sólo se pueden leer las informaciones en el orden inverso en que han sido escritas, siguiendo el principio de "el último en entrar, será el primero en salir", y de ahí que también se le llame memoria LIFO (*last-in, first-out*).

Las dos operaciones posibles con una pila son introducir ("push") y extraer ("pop") como indica la figura 4.2.

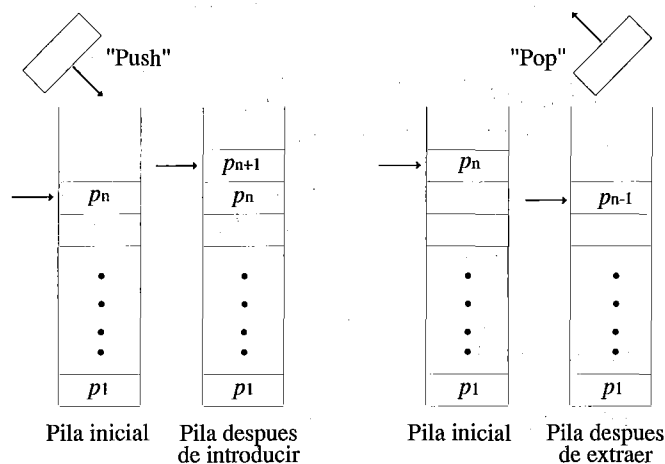


Figura 4.2.

El símbolo " \rightarrow " en la figura indica en cada momento cuál es la "cabeza" o "cima" de la pila ².

Un *autómata de pila*, AP, es un autómata finito al que se le añade una pila potencialmente infinita para guardar resultados intermedios. Puede considerársele como una MT con dos cintas y dos cabezas (figura 4.3). Pero no por ello es más potente que una MT. De hecho, su capacidad de procesamiento es inferior a la del ALL, debido a las siguientes restricciones sobre las posibles operaciones con las cintas:

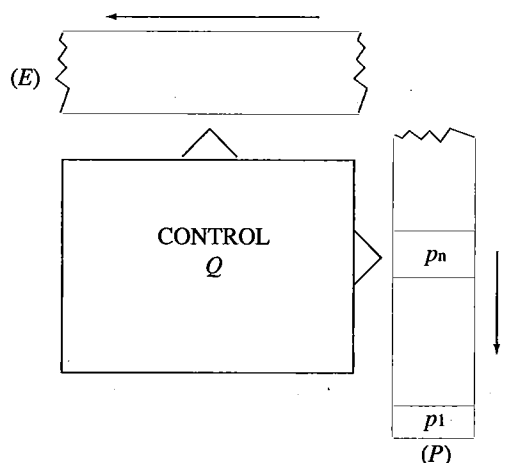


Figura 4.3.

La cinta (E) se *desplaza en solo sentido*, y la correspondiente cabeza *sólo puede leer*. En la cinta (P) (*limitada en un extremo*) puede leerse, en cuyo caso desaparece p_n y se desplaza hacia arriba (según la figura) o escribirse, para lo cual se desplaza hacia abajo y se introduce p_{n+1} sobre p_n .

Las operaciones elementales que puede realizar un AP son de dos tipos:

- Dependientes de E : se lee un $e_i \in E$, se desplaza la cinta (E), y, en función de e_i, q_j, p_k , el control pasa a otro estado q_ℓ y en la pila (P) se introduce p_{k+1} , o se extrae p_k , o no se hace nada;
- independientes de E : lo mismo, sólo que e_i no interviene y (E) no se mueve, lo que permite manejar la pila sin tener en cuenta las informaciones de entrada.

En cualquier caso, si se vacía la pila (es decir, se extrae p_1) el AP se para.

² Si la pila se simula en la memoria central de un ordenador mediante un programa, existirá un puntero a la cima de la pila, es decir, una variable que se actualizará en cada operación para que su valor sea la dirección de la cima.

4.2 Lenguaje aceptado por un reconocedor de pila

El reconocedor de pila, R_{AP} , será un AP inicializado en un estado q_1 y con p_1 como única información en la cinta (P). La aceptación de cadenas, y, consecuentemente, el *lenguaje aceptado* por R_{AP} , $L(R_{AP})$ se puede definir según dos criterios diferentes:

- *por pila vacía*: $x \in L(R_{AP})$ si al leer el último símbolo de x la pila queda vacía (y, por tanto, el AP se para);
- *por estados finales*: $x \in L(R_{AP})$ si al leer el último símbolo de x el control queda en un estado $q_A \in F_A$, siendo $F_A \subset Q$ un conjunto de estados definidos a priori como estados finales de aceptación.

En un estudio formalizado de los AP, que aquí no abordamos, se demuestra que *ambas definiciones de $L(R_{AP})$ son equivalentes*, en el sentido de que la clase de los lenguajes aceptados por las estructuras del tipo AP es la misma en ambos casos.

4.3 Teorema AP1

Para toda gramática libre de contexto, $G2$, existe un reconocedor de pila, R_{AP} , tal que $L(R_{AP}) = L(G2)$.

4.4 Teorema AP2

Para todo reconocedor de pila, R_{AP} , existe una gramática libre de contexto, $G2$, tal que $L(G2) = L(R_{AP})$.

4.5 Conclusión

De los teoremas enunciados se desprende que

$$\{ L(R_{AP}) \} = \{ L(G2) \}$$

5. Lenguajes regulares y autómatas finitos

5.1 Autómata finito no determinista

En el tema "Autómatas" se han estudiado los reconocedores finitos y se ha visto que la clase de lenguajes que pueden reconocer son los lenguajes regulares. Aquí vamos a demostrar que esa clase de lenguajes coincide precisamen-

te con la clase de lenguajes que pueden ser generados por gramáticas de tipo 3 ó regulares. Para ello tenemos que recurrir a un concepto auxiliar: el AF no determinista. Su necesidad aparece claramente teniendo en cuenta cómo se establecen las relaciones entre gramáticas de tipo 3 y autómatas. En efecto, si en un AF existe una transición del estado A al estado B bajo el efecto de la entrada e , veremos que la gramática correspondiente contiene la producción $A \rightarrow eB$, y a la inversa. Ahora bien, nada impide que una gramática regular contenga simultáneamente las reglas $A \rightarrow eB$ y $A \rightarrow eC$, lo que nos lleva a considerar la posibilidad de que *del estado A con entrada e se pase bien al estado B , bien al C* . Un AF en el que tal cosa puede ocurrir se llama *no determinista*. Quede bien claro que no es la idea de probabilidad la que aquí interviene. El AF no determinista debe ser considerado con un concepto abstracto, desprovisto de interpretación física. (El autómata estocástico estudiado en el tema "Autómatas", capítulo 5 apartado 3, podría verse como un autómata no determinista si atribuimos probabilidades de manera consistente a las producciones.)

Definición 5.1.1.

Un autómata finito no determinista es una quintupla

$$\text{AFND} = \langle E, S, Q, f, g \rangle,$$

donde todo es igual que en un autómata finito (tema "Autómatas" capítulo 2, apartado 1.1), *salvo que* el rango de la función de transición no es Q , sino $P(Q)$ (siendo $P(Q)$ el conjunto de las partes de Q):

$$f: E \times Q \rightarrow P(Q)$$

Es decir, $f(e, q) = \{q_a, q_b, \dots, q_p\} \subset Q$. (Obsérvese que puede ser $f(e, q) = \emptyset$.)

Un reconocedor finito no determinista se define siguiendo la misma línea que en el caso determinista: se considera un conjunto de estados finales, F , un estado inicial, q_1 , y

$$\text{RFND} = \langle E, Q, f, q_1, F \rangle$$

El dominio de la función de transición puede extenderse a $E^* \times Q$: Recuérdese que en el caso determinista hacíamos (tema "Autómatas", capítulo 2, apartado 3.2) $f(\lambda, q) = q$; $f(x_1x_2, q) = f(x_2, f(x_1, q))$; ahora haremos

$$f(\lambda, q) = \{q\}; f(x_1x_2, q) = \bigcup_{\substack{q_k \text{ en} \\ f(x_1, q)}} f(x_2, q_k)$$

También puede extenderse tal dominio a $E^* \times P(Q)$; para ello haremos:

$$f(x, \emptyset) = \emptyset$$

$$f(x, \{q_a, q_a, \dots, q_\ell\}) = \bigcup_{j=a}^{\ell} f(x, q_j)$$

Ejemplo 5.1.2.

Sea el RFND definido por

$$E = \{a, b\}; Q = \{q_1, q_2, q_3, q_4\}; F = \{q_4\}$$

y la función de transición de la figura 4.4

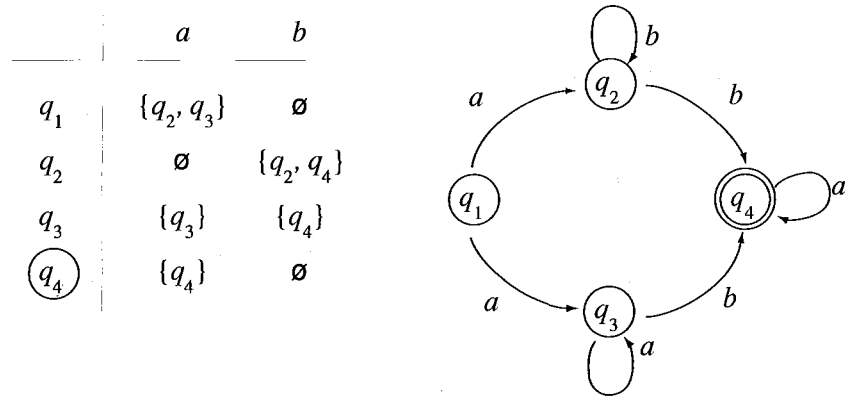


Figura 4.4.

Estudiemos, por ejemplo, las transiciones producidas por la cadena $x = abbab$

$$\begin{aligned}
 f(a, q_1) &= \{q_2, q_3\} \\
 f(ab, q_1) &= f\{b, q_2\} \cup f\{b, q_3\} = \{q_2, q_4\} \\
 f(abb, q_1) &= f\{b, q_2\} \cup f\{b, q_4\} = \{q_2, q_4\} \\
 f(abba, q_1) &= f\{a, q_2\} \cup f\{a, q_4\} = \{q_4\} \\
 f(abbab, q_1) &= f\{a, q_4\} = \emptyset
 \end{aligned}$$

Si quisiéramos calcular, por ejemplo, $f(a, \{q_1, q_4\})$, teniendo en cuenta cómo se ha definido la extensión a $E^* \times P(Q)$, tendríamos:

$$f(a, \{q_1, q_4\}) = f(a, q_1) \cup f(a, q_4) = \{q_2, q_3, q_4\}$$

5.2 Lenguaje aceptado por un reconocedor finito no determinista, y equivalencia con algún reconocedor finito determinista

Definiremos el *lenguaje aceptado por un RFND* como el conjunto de cadenas para las cuales la función de transición conduce a un subconjunto de Q dentro del cual se encuentra al menos un estado final:

$$L(\text{RFND}) = \{ x \in E^* \mid [f(x, q_1) = \{ q_a, q_b, \dots, q_\ell \}] \wedge [\exists q_i (i = a, \dots, \ell) \in F] \}$$

Así, en el ejemplo anterior vemos que la cadena a es rechazada, ab , abb , y $abba$ son aceptadas, y $abbab$ es rechazada. Es fácil ver que el lenguaje admite una expresión regular: ab^*ba^* es la expresión regular del conjunto de cadenas que, pasando por el estado q_2 , son aceptadas, mientras que aa^*ba^* es la de las que pasan por q_3 ; por tanto, la expresión regular de todas será: $ab^*ba^* + aa^*ba^* + aa^*ba^* = a(a^* + b^*)ba^*$.

Pasaremos ahora a ver que la clase de los lenguajes aceptados por reconocedores finitos no deterministas coincide con la de los aceptados por reconocedores finitos deterministas. Para ello será preciso demostrar dos cosas:

- que, dado un RF (determinista), siempre existe un RFND tal que $L(\text{RFND}) = L(\text{RF})$, (y, por tanto, $\{L(\text{RF})\} \subset \{L(\text{RFND})\}$), y

- que, dado un RFND, siempre existe un RF tal que $L(\text{RF}) = L(\text{RFND})$ (y, por tanto, $\{L(\text{RFND})\} \subset \{L(\text{RF})\}$), que, con lo anterior, demostrará que $\{L(\text{RFND})\} = \{L(\text{RF})\}$.

Lo primero es evidente, ya que un RF es un caso particular de RFND en el que los elementos del rango de f son subconjuntos unitarios de $P(Q)$. Lo segundo no es tan evidente, por lo que pasaremos a enunciarlo en forma de teorema y demostrarlo.

Teorema 5.2.1.

Para lo reconocedor finito no determinista, $\text{RFND} = \langle E, Q, f, q_1, F \rangle$, puede construirse un reconocedor finito determinista, $\text{RF} = \langle E, Q', f', q'_1, F' \rangle$, tal que $L(\text{RF}) = L(\text{RFND})$.

Hagamos: $Q' = P(Q)$ (de modo que RF tendrá, en general, $\text{card}(Q') = 2^{\text{card}(Q)}$ estados). Al estado de Q' que corresponda a $\{q_a, q_b, \dots, q_\ell\}$ lo denotaremos $[q_a, q_b, \dots, q_\ell]$.

$$\begin{aligned} f'(e, [q_a, \dots, q_\ell]) &= [q_m, \dots, q_k] \quad \text{si y sólo si} \\ f(e, \{q_a, \dots, q_\ell\}) &= \{q_m, \dots, q_k\}. \end{aligned}$$

(Es decir, se calcula $f'(e, q')$ aplicando f a cada estado q de los que figuran en q' y haciendo la unión de todos los resultantes.)

$$\begin{aligned} q'_1 &= [q_1] \\ F' &= \{q' \in Q' \mid q_f \in q' \text{ y } q_f \in F'\} \end{aligned}$$

(Es decir, para q' sea estado final basta que uno o más de los estados de Q que lo componen sea final.)

Para demostrar que ambos autómatas aceptan el mismo lenguaje bastará comprobar que, para todo $x \in E^*$, $f'(x, q_1) \in F'$ si y sólo si $f(x, q_1)$ contiene un estado (o varios) $q_f \in F$, y, teniendo en cuenta la definición de F' , esto será evidentemente cierto si demostramos que

$$f'(x, q_1) = [q_a, \dots, q_\ell] \quad \text{si y sólo si} \quad f(x, q_1) = \{q_a, \dots, q_\ell\}$$

Tal demostración puede hacerse por inducción sobre la longitud de x : Para $\lg(x) = 0$, ($x = \lambda$) es inmediato, puesto que $f'(\lambda, q'_1) = q'_1 = [q'_1]$, y $f(\lambda, q_1) = \{q_1\}$. Supongamos que es cierto para $\lg(x) \leq 1$; entonces para $e \in E$,

$$f'(xe, q'_1) = f'(e, f'(x, q'_1))$$

Pero por la hipótesis de la inducción,

$$f'(x, q'_1) = [q_a, \dots, q_\ell] \quad \text{si y sólo si} \quad f(x, q_1) = \{q_a, \dots, q_\ell\}$$

y, por definición de f' ,

$$\begin{aligned} f'(e, [q_a, \dots, q_\ell]) &= [q_m, \dots, q_k] \quad \text{si y sólo si} \\ f(e, \{q_a, \dots, q_\ell\}) &= \{q_m, \dots, q_k\}. \end{aligned}$$

Por tanto,

$$f'(xe, q'_1) = [q_m, \dots, q_k] \quad \text{si y sólo si} \quad f(xe, q_1) = \{q_m, \dots, q_k\}$$

con lo que queda demostrado.

Ejemplo 5.2.2.

Tomemos el RFND del ejemplo 5.1.2. Siguiendo la construcción del Teorema 5.2.1, el RF tendrá, en principio, $2^4 = 16$ estados:

$$Q' = \{\emptyset, [q_1], [q_2], [q_3], [q_4], [q_1, q_2] \dots [q_1, q_2, q_3, q_4]\}$$

$$q'_1 = [q_1]$$

$$F' = \{[q_4], [q_1, q_4], [q_2, q_4], [q_3, q_4], [q_1, q_2, q_4], \dots [q_1, q_2, q_3, q_4]\}$$

y f' vendrá dada por la siguiente tabla:

		a	b
	\emptyset	\emptyset	\emptyset
	$[q_1]$	$[q_2, q_3]$	\emptyset
→	$[q_2]$	\emptyset	$[q_2, q_4]$
	$[q_3]$	$[q_3]$	$[q_4]$
	$[q_4]$	$[q_4]$	\emptyset
→	$[q_1, q_2]$	$[q_2, q_3]$	$[q_2, q_4]$
→	$[q_1, q_3]$	$[q_2, q_3]$	$[q_4]$
→	$[q_1, q_4]$	$[q_2, q_3, q_4]$	\emptyset
	$[q_2, q_3]$	$[q_3]$	$[q_2, q_4]$
	$[q_2, q_4]$	$[q_4]$	$[q_2, q_4]$
→	$[q_3, q_4]$	$[q_3, q_4]$	$[q_4]$
→	$[q_1, q_2, q_3]$	$[q_2, q_3]$	$[q_2, q_4]$
→	$[q_1, q_2, q_4]$	$[q_2, q_3, q_4]$	$[q_2, q_4]$
→	$[q_1, q_3, q_4]$	$[q_2, q_3, q_4]$	$[q_4]$
→	$[q_2, q_3, q_4]$	$[q_3, q_4]$	$[q_2, q_4]$
→	$[q_1, q_2, q_3, q_4]$	$[q_2, q_3, q_4]$	$[q_2, q_4]$

Ahora bien, en un RF los estados que no son accesibles desde el inicial pueden eliminarse. Así, eliminamos $[q_2]$ porque no lo vemos aparecer dentro de la tabla, y lo mismo $[q_1, q_2]$, etc. (los señalados con una flecha). Obsérvese que $[q_2, q_3, q_4]$ sólo es accesible desde otros que previamente han sido eliminados por lo que también puede eliminarse, y lo mismo ocurre con $[q_3, q_4]$. Naturalmente, el único del que no puede prescindirse en ningún caso es $[q_1]$. \emptyset en este caso, no puede eliminarse, ya que es accesible desde $[q_1]$ (y desde $[q_4]$). Finalmente, de los 16 estados nos hemos quedado con 6, y el resultado puede representarse en forma de diagrama de Moore según indica la figura 4.5.

Comprobemos que el lenguaje aceptado es el mismo del RFND de partida (expresión regular $a(a^* + b^*)ba^*$). Al estado final $[q_2, q_4]$ sólo podemos ir pasando por $[q_2, q_3]$; tenemos así que las cadenas aceptadas en $[q_2, q_4]$ tienen por expresión regular: abb^* ; a $[q_4]$ puede llegarse por $[q_2, q_4]$ (abb^*aa^*) o por $[q_3]$ (aaa^*ba^*).

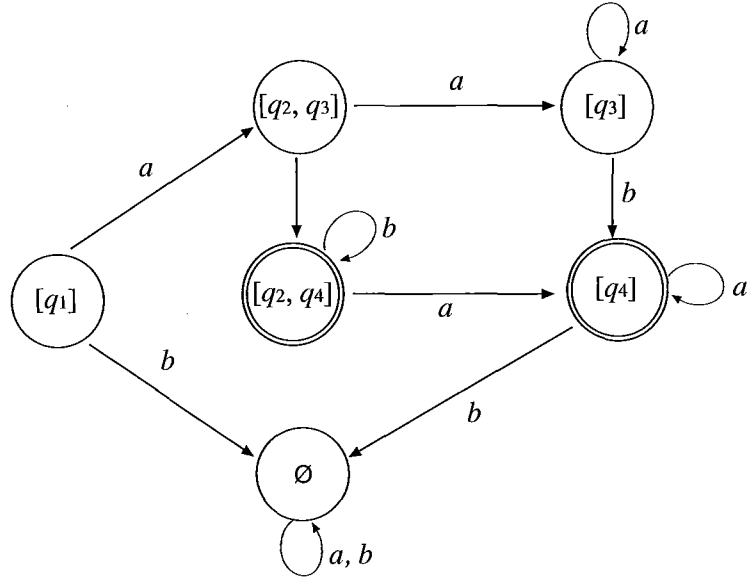


Figura 4.5.

Tenemos, pues, en total:

$$\begin{aligned}
 abb^* + abb^*aa^* + aaa^*ba^* &= abb^* (\lambda + aa^*) + aaa^*ba^* = \\
 &= abb^*a^* + aaa^*ba^* = ab^*ba^* + aaa^*ba^* = \\
 &= a (b^* + aa^*)ba^* = a(b^* + a^*)ba^*
 \end{aligned}$$

(Esta última igualdad se ha obtenido teniendo en cuenta que $b^* + aa^* = b^* + \lambda + aa^* = b^* + a^*$). Naturalmente, llegaríamos al mismo resultado aplicando el algoritmo general de análisis desarrollado en el tema "Autómatas", capítulo 4, apartado 4.

5.3 Teorema AF1

Para toda gramática regular, $G3$, existe un reconocedor finito, RF, tal que $L(\text{RF}) = L(G3)$.

Para demostrar este teorema construiremos un RFND que reconoce exactamente el lenguaje aceptado por una $G3$ dada, al que le corresponderá un RF de acuerdo con el teorema 5.2.1.

Sea $G3 = \langle E_A, E_T, P, S \rangle$ una gramática regular. Definimos RFND = $\langle E, Q, f, q_1, F \rangle$ del siguiente modo:

$$\begin{aligned}
E &= E_T = \{a_1, a_2, \dots, a_n\} \\
Q &= E_A \cup \{X\} = \{A_1, A_2, \dots, A_n, X\} \\
F &= \{X\} \\
q_1 &= S \\
\text{Si } (A_i \rightarrow a_j) \in P, &\text{ entonces } X \in f(a_j, A_i) \\
\text{Si } (A_i \rightarrow a_j A_k) \in P, &\text{ entonces } A_k \in f(a_j, A_i) \\
(\forall a_i \in E_T) & (f(a_i, X) = \emptyset)
\end{aligned}$$

Veamos ahora que $L(\text{RFND}) \equiv L(G3)$

a) Sea $x = a_1 a_2 \dots a_k \in L(G3)$; entonces, deberán existir

$$A_1, A_2, \dots, A_{k-1} \in E_A \text{ tales que}$$

$$S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \dots \Rightarrow a_1 a_2 \dots a_{k-1} A_{k-1} \Rightarrow a_1 a_2 \dots a_{k-1} a_k$$

y para ello P debe contener las reglas

$$\begin{array}{l}
S \rightarrow a_1 A_1 \\
A_1 \rightarrow a_2 A_2 \\
\vdots \\
\vdots \\
A_{k-1} \rightarrow a_k
\end{array}$$

Por la definición de f , $A_1 \in f(a_1, q_1)$; $A_2 \in f(a_2, A_1)$; ... $X \in f(a_k, A_{k-1})$; por consiguiente, $X \in f(a_1 a_2 \dots a_k, q_1)$, y, como X es el estado final, $a_1 a_2 \dots a_k \in L(\text{RFND})$. Esto nos dice que $L(G3) \subset L(\text{RFND})$.

b) A la inversa, si $x = a_1 a_2 \dots a_k \in L(\text{RFND})$, deberá existir una secuencia de estados $A_1, A_2, \dots, A_{k-1}, X$, tal que

$$\begin{array}{l}
A_1 \in f(a_1, q_1) \\
A_2 \in f(a_2, A_1) \\
\vdots \\
\vdots \\
A_{k-1} \in f(a_{k-1}, A_{k-2}) \\
X \in f(a_k, A_{k-1})
\end{array}$$

y, conforme a la construcción del RFND, $G3$ deberá tener las reglas $S \rightarrow a_1 A_1$; $A_1 \rightarrow a_2 A_2$; ... $A_{k-1} \rightarrow a_k$, con las cuales se podrá hacer la derivación $S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \dots \Rightarrow a_1 a_2 \dots a_k$, es decir, $a_1 a_2 \dots a_k \in L(G3)$. Es decir, $L(\text{RFND}) \subset L(G3)$.

En resumen, $L(\text{RFND}) = L(G3)$.

Ejemplo 5.3.1.

Tomemos la gramática regular del capítulo 2, apartado 4.5.

$$E_A = \{A, S\}; E_T = \{a, b\};$$

$$P = \left\{ \begin{array}{l} S \rightarrow aS \\ S \rightarrow aA \\ A \rightarrow bA \\ A \rightarrow b \end{array} \right\}$$

El correspondiente RFND

$$E = E_T = \{a, b\}; Q = \{A, S, X\}; F = \{X\}; q_1 = S$$

y la función de transición de la figura 4.6.

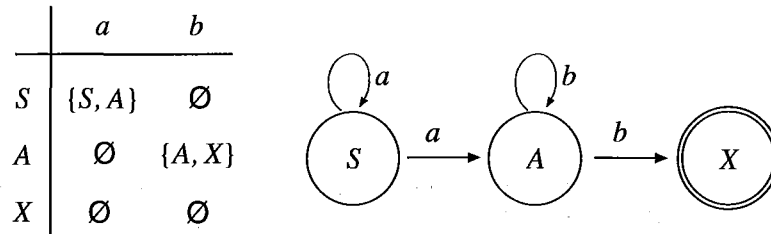


Figura 4.6.

Podemos construir un RF que acepte el mismo lenguaje, de acuerdo con el teorema 5.2.1:

	a	b
\emptyset	\emptyset	\emptyset
$[S]$	$[S, A]$	\emptyset
$\rightarrow [A]$	\emptyset	$[A, X]$
$\rightarrow [X]$	\emptyset	\emptyset
$[S, A]$	$[S, A]$	$[A, X]$
$\rightarrow [S, X]$	$[S, A]$	\emptyset
$[A, X]$	\emptyset	$[A, X]$
$\rightarrow [S, A, X]$	$[S, A]$	$[A, X]$

Eliminamos los estados $[A]$, $[X]$, $[S, X]$, $[S, A, X]$, inaccesibles desde $[S]$, y resulta el diagrama de Moore de la figura 4.7.

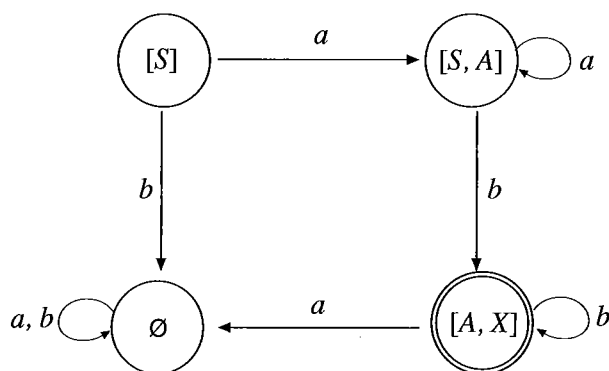


Figura 4.7.

Es fácil comprobar que el lenguaje viene dado por la expresión regular aa^*bb^* .

5.4 Teorema AF2

Para todo reconocedor finito, RF, existe una gramática regular, $G3$, tal que $L(G3) = L(RF)$.

Dado $RF = \langle E, Q, f, q_1, F \rangle$, construiremos $G3 = \langle E_A, E_T, P, S \rangle$ así:

$$E_A = Q; E_T = E; S = q_1$$

Si $f(a, q_i) = q_j$, entonces $q_i \rightarrow aq_j$

Si $f(a, q_i) = q_j$ y $q_j \in F$, entonces $q_i \rightarrow a$ (además de $q_i \rightarrow aq_j$)

Si $(\forall a) [(f(a, q_i) = q_j) \wedge (q_j \notin F)]$, entonces puede eliminarse q_i y todas las reglas en las que figure.

La demostración de que $L(G3) = L(RF)$ se hace viendo que $(q_1 \xrightarrow{*}_{G_3} x)$ si y sólo si $(\forall x \in E^*) [f(x, q_1) \in F]$, de manera similar a la del teorema anterior.

Ejemplo 5.4.1.

Consideremos el RF al que llegábamos en el ejemplo 5.3.1 y retrocedamos ahora en busca de la gramática. Haciendo, para simplificar la escritura,

$$[S] = S, [S, A] = A, [A, X] = B, \text{ tendremos:}$$

$$E_A = \{S, A, B\}; E_T = \{a, b\}$$

Obsérvese que no hemos incluido \emptyset en E_A porque

$$f(a, \emptyset) = f(b, \emptyset) = \emptyset.$$

Siguiendo las normas para la construcción de las reglas obtenemos:

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow aA \\ A &\rightarrow bB \\ A &\rightarrow b \\ B &\rightarrow bB \\ B &\rightarrow b \end{aligned}$$

Esta gramática tiene un símbolo auxiliar (B) y dos reglas más que aquella de la que habíamos partido, aunque, naturalmente, ambas deben ser equivalentes. La diferencia se ha originado en el paso a través del RF determinista. El lector puede extender la construcción del teorema AF2 al caso más general de RFND, y aplicarla al ejemplo 5.3.1 para ver que entonces sí se llega a la misma gramática original.

5.5 Conclusión

Según el teorema AF1, $\{L(G3)\} \subset \{L(RF)\}$, y según el AF2, $\{L(RF)\} \subset \{L(G3)\}$. Además, del tema "Autómatas" (capítulo 4, apartado 4.4.1; teorema de análisis) sabemos que $\{L(RF)\} = \{L_{\text{regulares}}\}$. En resumen:

$$\{L_{\text{regulares}}\} = \{L(RF)\} = \{L(G3)\}$$

5.6 Observaciones sobre la cadena vacía

En el teorema AF1 hemos supuesto implícitamente que $\lambda \notin L(G3)$.

Si $\lambda \in L(G3)$ entonces, según se vio en el capítulo anterior, existirá la regla $S \rightarrow \lambda$ (y S no aparecerá a la derecha de ninguna otra regla). En este caso, la única diferencia es que habrá que considerar S también como estado final $F = \{S, X\}$.

En cuanto al caso inverso (teorema AF2), si en el RF de partida $q_1 \in F$, entonces $\lambda \in L(RF)$, y $L(G3) = L(RF) - \lambda$. Según vimos, puede construirse $G'3$ tal que

$$L(G'3) = L(G3) \cup \{\lambda\} = L(RF)$$

6. Jerarquía de autómatas

Paralelamente a la jerarquía de lenguajes, aparece una jerarquía de autómatas, desde la MT hasta el AF. Para compararlos puede utilizarse como criterio la capacidad de memoria de cada uno.

La MT dispone, sobre la cinta, de una capacidad ilimitada de casillas para memorizar una cantidad ilimitada de símbolos.

La cinta de un ALL es también ilimitada, pero para una determinada cadena de entrada, x , de longitud $\ell = \lg(x)$, sólo puede utilizar ℓ casillas. Además, habrá que añadir a esta capacidad variable de memoria (dependiente, en cada caso, de cada entrada), una capacidad fija, n , que corresponde a la capacidad de la unidad central para memorizar estados. En resumen, la memoria de un ALL es una función de la longitud ℓ de la cadena de entrada: $\ell + n$. La forma lineal de esta función justifica el nombre del autómata.

El AP sólo puede utilizar la cinta de entrada para leer, pero dispone de una cinta de trabajo o pila potencialmente ilimitada que le sirve de memoria. Aquí también puede estimarse, para una longitud ℓ de la cadena de entrada, un límite superior de la memoria a utilizar. En efecto, si la cadena más larga que puede escribirse en la pila como consecuencia de la lectura de un símbolo de entrada tiene longitud k , la capacidad total de memoria será la suma de la parte variable (pila): $k\ell$ y de la parte fija, n ; en total, $k\ell + n$. Esta es también una función lineal de ℓ , por lo que del AP puede también decirse que es un autómata limitado linealmente, pero con dos restricciones importantes: la cinta de entrada sólo se desplaza en un sentido, y la memoria tiene estructura de pila (si se quiere recuperar un símbolo que no está en la cima, hay que borrar toda la información comprendida entre la cima y el símbolo, cosa que no ocurre con la MT ni el ALL).

La capacidad de memoria de un AF es fija e independiente de la cadena de entrada.

Finalmente, es preciso que mencionemos un punto importante que, al omitir la demostración de la mayoría de los teoremas, hemos pasado por alto: que los autómatas de que venimos hablando son, en general, no deterministas, es decir, que la función de transición de la unidad de control tiene la forma que veíamos en el AF no determinista. Esto solamente serviría para demostrar los teoremas, y no tendría mayor importancia, si en todos los tipos de reconocedores ocurriese lo que en el RF: que para todo RFND puede encontrarse un RF que acepte el mismo lenguaje, pero no es así:

a) En la MT, se demuestra que $\{L(R_{MT})\} = \{L(R_{MTND})\}$, es decir, ocurre lo mismo que en el RF.

b) En el ALL *no se sabe si* $\{L(R_{ALL})\} = \{L(R_{ALLND})\}$ o $\{L(R_{ALL})\} \subset \{L(R_{ALLND})\}$. Por tanto, cuando se habla de equivalencia de lenguajes sensibles al contexto y lenguajes aceptados por R_{ALL} debe entenderse que éstos

son *no deterministas*. Por supuesto, para todo R_{ALLND} puede diseñarse un R_{MT} que acepte el mismo lenguaje; lo único que ocurre es que la longitud de cinta necesaria en el R_{MT} (determinista) puede ser una función exponencial de la longitud de la cadena de entrada, en lugar de lineal, como en el R_{ALLND} .

c) En el AP, se sabe que $\{L(R_{AP})\} \subset \{L(R_{APND})\}$, y, es más, se conoce el tipo de gramáticas correspondientes a los R_{AP} deterministas, que son una particularización del tipo general de gramáticas libres de contexto, llamadas *gramáticas deterministas*, y que, entre otras propiedades, no son ambiguas.

7. Resumen

La figura 4.8 resume las principales conclusiones de este capítulo.

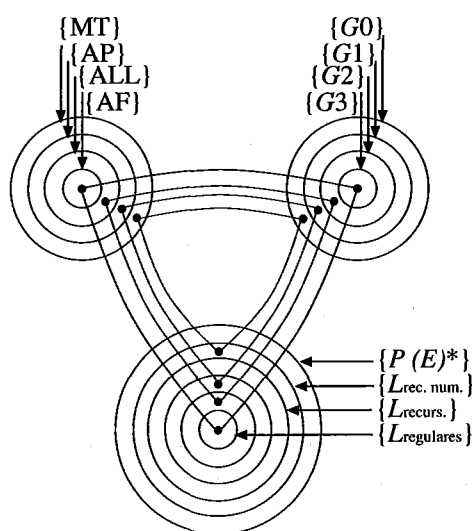


Figura 4.8.

8. Notas histórica y bibliográfica

En los temas "Autómatas" y "Algoritmos" se han dado referencias históricas sobre los AF y las MT, respectivamente. El concepto de AF no determinista se debe a Rabin y Scott (1959), aunque las relaciones entre AF y gramáticas regulares ya habían sido establecidas por Chomsky y Miller (1958). La

equivalencia entre lenguajes aceptados por MT y lenguajes de tipo 0 fue demostrada por Chomsky (1959).

El nombre y el concepto de ALL fueron introducidos por Myhill (1960). La generalización al caso no determinista y la equivalencia con las gramáticas sensibles al contexto se deben a Kuroda (1964).

Oettinger (1961) fue el primero en definir formalmente el AP, y Chomsky (1962) y Evey (1963), independientemente, demostraron su relación con los lenguajes libres de contexto.

Como bibliografía de consulta son recomendables los mismos libros citados en los anteriores capítulos, y, especialmente, los de Hopcroft y Ullman (1969, 1979).

9. Ejercicios

9.1 Hallar una gramática regular que genere el lenguaje correspondiente al RF del ejemplo 5.2.2. Partir del RFND del ejemplo 5.1.2 y obtener otra gramática, equivalente pero más sencilla.

9.2 Hallar gramáticas regulares correspondientes a los cuatro ejemplos que se utilizaron en el capítulo 4 del tema "Autómatas".

9.3 Dada una expresión regular

$$\alpha = 00^* + 10^*10^*$$

obtener un reconocedor finito del correspondiente lenguaje, y una gramática regular que lo genere.

9.4 Dada la gramática regular definida por:

$$E_A = \{S, A, B, C\}; E_T = \{0, 1\};$$

$$P = \left\{ \begin{array}{ll} S \rightarrow 1A; & A \rightarrow 1 \\ S \rightarrow 1B; & B \rightarrow 1A \\ A \rightarrow 0A; & B \rightarrow 1C \\ A \rightarrow 0C; & B \rightarrow 1 \\ A \rightarrow 1C; & C \rightarrow 1 \end{array} \right\}$$

dar una expresión regular del lenguaje generado por ella y diseñar un circuito secuencial con biestables JK que sirva como reconocedor de tal lenguaje.

9.5 Repetir el ejercicio anterior para la gramática

$$E_A = \{S, A\}; \quad E_T = \{a, b, c, d\};$$

$$P = \left\{ \begin{array}{ll} S \rightarrow aS; & S \rightarrow b \\ S \rightarrow aA; & S \rightarrow c \\ S \rightarrow bA; & S \rightarrow d \\ S \rightarrow cA; & A \rightarrow aA \\ S \rightarrow dA; & A \rightarrow bA \\ S \rightarrow a; & A \rightarrow a \\ & A \rightarrow b \end{array} \right\}$$

9.6 (Hopcroft y Ullman, 1969). Utilizar el concepto de RFND para demostrar que, si L es un lenguaje regular,

$$L^R = \{x \mid x^R \in L\}$$

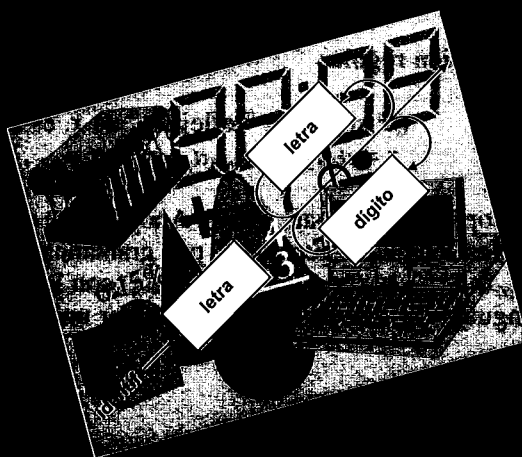
es también regular

(x^R significa "reflejada" de x , es decir si
 $x = a_1a_2, \dots, a_{n-1}a_n$, $x^R = a_na_{n-1}, \dots, a_2a_1$).

9.7 (Hopcroft y Ullman, 1969). Apoyándose en el ejercicio anterior, demostrar que los lenguajes generados por gramáticas cuyas reglas son de la forma $A \rightarrow Ba$, ó $A \rightarrow a$ ($a \in E_T$; $A, B \in E_A$) son lenguajes regulares (y viceversa: todo lenguaje regular puede generarse por una gramática de ese tipo).

Fundamentos de informática

5



5

Aplicaciones a los lenguajes de programación

1. Lenguajes, traductores e intérpretes

En los tres capítulos anteriores hemos presentado los fundamentos de la teoría de lenguajes formales y su relación con la teoría de autómatas. Veremos ahora cómo estos conceptos teóricos se aplican en la práctica a la problemática de la programación de ordenadores. El asunto es demasiado complejo y extenso como para pretender darle aquí un tratamiento completo. Por otra parte, es bastante probable que el lector tenga algunos (o muchos) conocimientos previos sobre programación de ordenadores. Por todo ello, nos limitaremos a apuntar las principales ideas, remitiendo a las orientaciones bibliográficas del apartado 7 a quien desee profundizar en alguno de los aspectos comentados.

Si L es un lenguaje de programación, $x \in L$ será un programa escrito en ese lenguaje, programa que corresponderá a algún algoritmo (tema "Algoritmos", capítulo 2) y que se habrá escrito para resolver algún problema. Entonces, lo que interesa es disponer de una máquina que *ejecute* x . Llamaremos *intérprete* de L a una máquina, M , que reconoce si $x \in L$, y, en caso afirmativo, lo ejecuta. Y diremos que L es el *lenguaje de máquina* de M .

Ahora bien, si afrontamos lo que ocurre en la realidad, nos encontramos con que los lenguajes de máquina de los ordenadores resultan prácticamente inutilizables para su uso directo. Codificar los algoritmos en un lenguaje cuyos únicos símbolos son "0" y "1" sería algo excesivamente tedioso para el programador, y de un coste prohibitivo por la cantidad de trabajo necesario, no sólo para la escritura de los programas, sino también para su depuración y mantenimiento. Por eso, desde los tiempos de los primeros ordenadores, uno de los campos de mayor actividad en informática ha sido el relacionado con el diseño de lenguajes que hagan más fácil y más productiva la tarea de programar. Escrito un programa en uno de estos lenguajes, $L1$, antes de que pueda ejecutarse en una máquina, $M2$, cuyo lenguaje de máquina es $L2$, es preciso traducirlo. Un *traductor* de $L1$ a $L2$, $T12$, es un programa que recibe como dato de entrada $x \in L1$ y, tras comprobar que, efectivamente, $x \in L1$, produce como resultado $y \in L2$. Naturalmente, si este traductor ha de ejecutarse en $M2$ deberá estar escrito en $L2$ (o bien, haber sido traducido previamente a $L2$ desde el lenguaje en que fue escrito). El conjunto formado por $T12$ y $M2$ puede entonces verse como una "máquina virtual" (capítulo 1, apartado 1), $M1$, cuyo lenguaje de máquina es $L1$. Podemos, por tanto, decir que $M1$ es un intérprete (reconocedor y ejecutor) de $L1$.

De acuerdo con la descripción que acabamos de hacer de la máquina virtual $M1$, la interpretación y de un programa $x \in L1$ se lleva a cabo en dos pasos: en el primero, la máquina real, $M2$, interpreta el programa traductor $T12$ dando como resultado de la ejecución $y \in L2$; después, la misma $M2$ interpreta y . A $L1$ se le llama *lenguaje fuente* y a $L2$ *lenguaje objeto*.

Pero hay otra alternativa para la ejecución en $M2$ de un algoritmo descrito por un programa $x \in L1$. Como sabemos (tema "Algoritmos", capítulo 3, apartado 2.2), x consta de una secuencia finita de instrucciones, instrucciones que tienen sentido para $M1$ (máquina virtual) pero no para $M2$ (máquina real). Podemos pensar en un programa para $M2$, $I12$, que vaya reconociendo, *una a una*, las instrucciones de x y haciendo que $M2$ ejecute las acciones oportunas para que dé el mismo resultado que hubiera dado $M1$. Este tipo de programa se llama *intérprete*.²

La figura 5.1 ilustra la descripción que acabamos de hacer sobre el funcionamiento de un traductor y el de un intérprete. Ambos son *procesadores* de lenguaje, y, desde el punto de vista del programador, ambos permiten disponer

¹ Utilizaremos el término "intérprete" con dos sentidos. Uno es el que acabamos de definir: una máquina que reconoce las órdenes o instrucciones de un programa escrito en su lenguaje de máquina y las ejecuta, una tras otra. El otro sentido, que veremos enseguida, y que es el más utilizado en la práctica, se refiere a un programa que va examinando y reconociendo una tras otra las instrucciones del programa original y generando las instrucciones oportunas para la máquina ejecutora.

² Este es el segundo sentido de "intérprete" que mencionábamos en la nota anterior. Obsérvese que, funcionalmente, la diferencia entre un programa traductor y un programa intérprete es la misma que hay entre las profesiones de traductor e intérprete.

de una máquina virtual $M1$, con lenguaje de máquina $L1$ (en el que el programador escribe sus programas) a partir de una máquina real $M2$, con lenguaje de máquina $L2$ (en el que sería mucho más difícil, o prácticamente absurdo, escribir los programas).

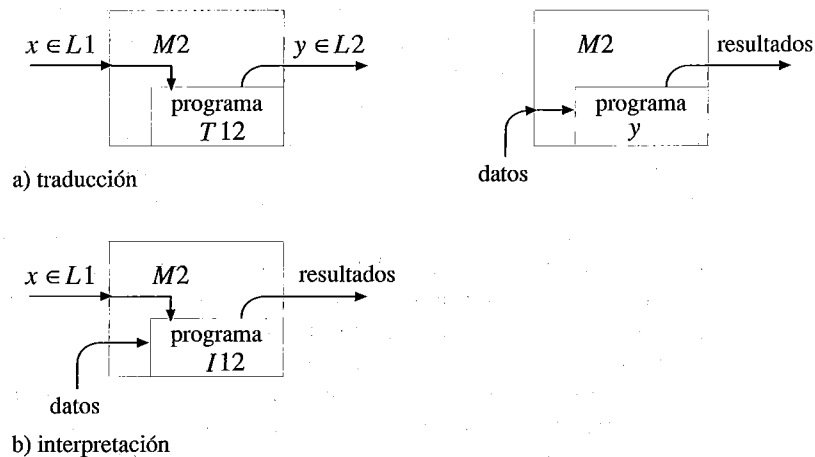


Figura 5.1.

Desde un punto de vista pragmático, hay diferencias importantes entre procesar un programa escrito en lenguaje fuente mediante un traductor o mediante un intérprete. Como el traductor procesa todo el programa fuente, $x \in L1$, antes de que $M2$ ejecute el programa objeto, $y \in L2$, cualquier error que se observe durante la ejecución de y obliga a modificar x y volver a traducirlo (mediante una nueva ejecución de $T12$). Por el contrario, si se usa un intérprete, las modificaciones pueden hacerse "sobre la marcha" y de modo interactivo entre el programador y el ordenador. El principal inconveniente del intérprete es que, para el mismo programa fuente y el mismo ordenador, la ejecución del programa es más lenta. Es fácil comprender por qué es así: si hay varias instrucciones dentro de un bucle, que van a ejecutarse muchas veces, el intérprete tiene que realizar con ellas las mismas operaciones de reconocimiento y de creación de instrucciones para $M2$ una y otra vez (cada vez que se las encuentra en el bucle), mientras que el traductor las habría pasado de una vez por todas al lenguaje objeto.

2. Lenguajes de programación

2.1 Niveles de lenguajes

Salvo para arquitecturas muy especiales, que son actualmente objeto de investigación, todos los lenguajes de máquina son muy parecidos, si abstraemos los detalles concretos de cada ordenador. En este sentido, podemos hablar de un "lenguaje de máquina" en sentido genérico. Pues bien, imaginemos una escala arbitraria que abarcara desde el lenguaje de máquina hasta el lenguaje natural sobre la cual pudiésemos situar cualquier lenguaje: cuanto más cercano esté del lenguaje natural más fácil será escribir programas (mejorando con ello la productividad y la calidad del software), pero, al mismo tiempo, más complejo el procesador (traductor o intérprete) y, normalmente, menos eficiente el programa objeto. Esto último se debe a que cuanto más alejado está el lenguaje fuente del lenguaje objeto tanto más difícil es conseguir que el programa traducido esté optimizado (o sea, que se ejecute utilizando el mínimo de recursos de la máquina y en el menor tiempo posible). El diseño de un lenguaje de programación implica, por tanto, un compromiso entre facilidad de uso para el programador y posibilidad de realizar un procesador eficiente.

La "escala arbitraria" que hemos sugerido sólo es una ilustración, porque no existe una métrica para situar a los lenguajes en ella. Para empezar, ¿qué es "cercanía al lenguaje natural"? Depende de quién vaya a hacer uso del lenguaje: un lenguaje como FORTRAN, por ejemplo, orientado a cálculos y evaluaciones de fórmulas algebraicas, es mucho más asequible para científicos e ingenieros que para quien se dedica al tratamiento de transacciones comerciales. No obstante, es tradicional distinguir dos zonas en esa escala imaginaria: la de los llamados *lenguajes de bajo nivel*, más cercanos al lenguaje de la máquina y la de los conocidos como *lenguajes de alto nivel*. Por encima de éstos podemos situar a los *lenguajes declarativos*.

Bajo la etiqueta de "lenguajes de bajo nivel" suelen englobarse, aparte de los lenguajes de máquina, los lenguajes *ensambladores* y *macroensambladores*. En un lenguaje ensamblador, cada instrucción no es más que la codificación simbólica de una instrucción del lenguaje de máquina. Por tanto, cada ordenador tiene su lenguaje de máquina y su lenguaje ensamblador. Ello implica que el programador está obligado a conocer con bastante detalle las peculiaridades de la máquina: número y longitud de los registros, modos de direccionamiento, formatos de representación interna, etc. Los traductores para lenguajes ensambladores se llaman *ensambladores*.

Los lenguajes llamados de alto nivel son los más utilizados: FORTRAN, BASIC, COBOL, PL/1, Pascal, C, Ada, etc. En principio, son independientes de la estructura de la máquina en la que se ejecutan los programas. Por ello, se dice que éstos son *transportables*: un programa escrito en un lenguaje de alto

nivel para un ordenador puede ejecutarse en otro cualquiera (siempre que se disponga del traductor o del intérprete adecuado). Los traductores para lenguajes de alto nivel se llaman *compiladores*, y, como veremos en el apartado 4, son bastante más complejos que los ensambladores.

Veamos con algunos ejemplos las características básicas de los distintos niveles de lenguajes.

2.2 Lenguajes de bajo nivel

2.2.1 Un lenguaje de máquina

Para ilustrar con un ejemplo muy sencillo cómo es un lenguaje de máquina y un lenguaje ensamblador y la programación con ellos, consideremos un modelo de ordenador con longitud de palabra³ de dieciséis bits, con un solo registro de dieciséis bits llamado acumulador y en el que las instrucciones de máquina son también de dieciséis bits. Las instrucciones pueden hacer referencia a una dirección de memoria para operar con su contenido. En nuestro modelo no consideraremos más que un modo de direccionamiento, el "directo". Concretamente, de los dieciséis bits de cada instrucción, los cuatro primeros corresponden al código de operación (por lo que puede haber $2^4 = 16$ operaciones diferentes) y los doce siguientes a una dirección de memoria (pueden direccionarse, así, $2^{12} = 4096$, o 4K palabras de la memoria). La dirección es, pues, un número entero comprendido entre 0 y 4095. Algunos de los códigos de operación son:

- **0000:** parar la ejecución (los doce bits de dirección se ignoran);
- **0001:** cargar en el acumulador, es decir, llevar el contenido (los dieciséis bits) de la palabra de memoria direccionada en el campo de dirección (los doce bits que siguen a 0001) al acumulador, borrando lo que previamente hubiera en éste;
- **0010:** almacenar el acumulador, es decir, llevar su contenido (los dieciséis bits) a la palabra de memoria direccionada, sin que se modifique lo que hay en el acumulador;
- **0011:** sumar al acumulador, es decir, sumar su contenido con el de la palabra de memoria direccionada y dejar el resultado en el mismo acumulador;

³ Suponemos que el lector tiene algunos conocimientos de la arquitectura básica de los ordenadores para entender este ejemplo. Así, debe saber que una "palabra" es el conjunto de bits que se transfieren en paralelo entre la memoria y la unidad de procesamiento.

- **0100:** restar del contenido del acumulador el de la palabra direccionada y dejar el resultado en el acumulador;
- **0101:** multiplicar el contenido del acumulador por el de la palabra direccionada y dejar el resultado en el acumulador.

Escribamos un programa en lenguaje de máquina para realizar la operación $A*(B+C)$, donde A , B y C son datos numéricos que el programa debe leer de algún periférico de entrada. Entre los códigos de operación que acabamos de definir no hemos incluido operaciones de comunicación con los periféricos, aunque, naturalmente, todo ordenador debe disponer de ellas; el hacerlo nos habría obligado a entrar en demasiados detalles, porque lo que el ordenador comunica con los periféricos son códigos de control y de caracteres, y es preciso utilizar programas de conversión de caracteres a números en binario, y viceversa. Por tanto, obviando este aspecto, supongamos que los números A , B y C se han leído de algún modo y han quedado almacenados en la memoria en las palabras de dirección 0, 1 y 2, respectivamente. El "algoritmo" consistirá en llevar B al acumulador (cargar el contenido de la palabra de memoria de dirección 1), sumarle C (contenido de 2) y lo que resulte multiplicarlo por A (contenido de 0), con lo que quedará en el acumulador $A*(B+C)$, que podemos almacenar, por ejemplo, en la dirección 3 (para, posteriormente, con otras instrucciones, escribir el resultado por un periférico de salida). El programa en lenguaje de máquina sería:

```

...
...
0001000000000001
0011000000000010
0101000000000000
0010000000000011
...
...
0000000000000000

```

Las dos líneas de puntos iniciales indican que habrá unas instrucciones de entrada de datos para leer los números, convertirlos a binario y guardarlos en las direcciones 0, 1 y 2. A continuación vienen cuatro instrucciones, cada una de dieciséis bits. La primera tiene código de operación 0001 (cargar) y el número 1 en su campo de dirección; es decir, su ejecución provocará que el contenido de esa palabra de memoria se transfiera al acumulador. El código de operación de la segunda instrucción es 0011 (sumar) y su campo de dirección es 2 (10 en binario); por tanto, cuando la máquina la ejecute, su efecto será el de sumar el contenido de la palabra de dirección 2 a lo que en ese momento

haya en el acumulador, y dejar la suma en el mismo acumulador. Con la siguiente, se multiplica ese resultado parcial por lo que haya en la dirección 0, y con la cuarta el resultado final se transfiere a la memoria, a la palabra de dirección 3 (11 en binario). A continuación se incluirían las instrucciones para sacar ese resultado por algún periférico, y, finalmente, viene la instrucción de parar.

Los ordenadores son máquinas de *programa almacenado*. Esto quiere decir que la secuencia de instrucciones que constituyen un programa se carga en la memoria y luego se ejecutan una detrás de otra, salvo cuando aparece una *instrucción de salto (o bifurcación)*. Se llaman así aquellas que provocan una "ruptura de secuencia" en la ejecución del programa, es decir, que la siguiente instrucción a ejecutar no es la que viene a continuación (almacenada en la siguiente palabra de la memoria), sino la que está almacenada en la dirección indicada por la propia instrucción de salto. Estas instrucciones suelen utilizarse para controlar bucles o acciones condicionales. Por ejemplo, nuestro ordenador puede tener una instrucción de *salto incondicional*, con código 0110, y otra de *salto si acumulador es cero* (0111). La primera, al ejecutarse, hace que la siguiente instrucción a ejecutar sea la que está almacenada en la dirección especificada en su campo de dirección, y la segunda lo mismo pero condicionado al hecho de que el contenido del acumulador esté formado por dieciséis ceros, de lo contrario, se ejecuta la siguiente de la sentencia.⁴ Obsérvese que esto implica que el programador tiene que conocer exactamente las direcciones en que van a almacenarse las instrucciones de su programa. Otra instrucción interesante es la de *salto a subprograma* (con código, por ejemplo, 1000), cuyo efecto es el mismo del salto incondicional, pero guardando en algún sitio (en otro registro, o en una determinada palabra de la memoria) la *dirección de retorno*, dirección de la instrucción que le sigue en la secuencia; el *subprograma* es un conjunto de instrucciones que termina con una *instrucción de retorno*, y es muy útil cuando han de realizarse repetidamente ciertas operaciones a lo largo de un programa (por ejemplo, cálculo de la raíz cuadrada, lectura de datos, etc.).

Veamos un ejemplo de programación en este lenguaje de máquina haciendo uso de las instrucciones de salto. Se leen dos números, A y B (suponemos que $A < B$) y se trata de igualar el primero con el segundo a base de sumarle sucesivamente una unidad, como indica el organigrama de la figura 5.2. Supondremos que a partir de la dirección 20 (10100 en binario) hay almacenado un subprograma que permite leer números. La primera vez que "se le llama" (con la instrucción de salto a subprograma) lee un número y lo almacena en la dirección 0; la siguiente, lee otro y lo almacena en la dirección 1, y así sucesivamente. Nuestro programa puede ser el siguiente (la primera columna indica la dirección de memoria, en decimal):

⁴ Suponemos que la representación del número "0" en esta máquina se hace mediante dieciséis ceros.

	contenido		
dir.	(binario)		comentario
0	_____		aquí se almacenará el primer dato
1	_____		aquí, el segundo
2	0000000000000001		esta es la constante 1
3	1000000000010100		salto al subprograma de lectura (lee un dato y lo almacena en la dirección 0)
4	1000000000010100		salto al subprograma de lectura (lee otro dato y lo guarda en 1)
5	0001000000000000		carga el primer número (A)
6	0100000000000001		le resta el segundo (B)
7	0111000000001100		si A=B, salta a ejecutar la instrucción almacenada en la dirección 12
8	0001000000000000		si no, carga A (almacenado en 0)
9	0011000000000010		le suma una unidad (está en 2)
10	0010000000000000		almacena el resultado en la dirección de A (dirección 0)
11	0110000000000101		y vuelve a la instrucción almacenada en la dirección 5
12	0000000000000000		cuando A=B se para

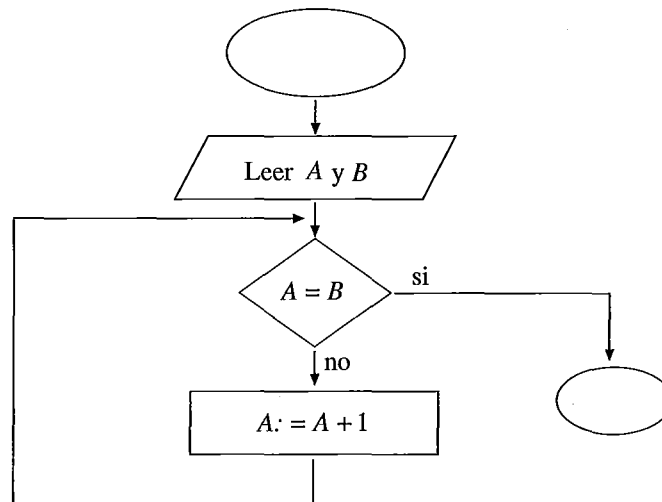


Figura 5.2.

2.2.2 Un lenguaje ensamblador

En un lenguaje ensamblador no hay que codificar en binario, porque a cada código de operación se le asigna un nombre nemotécnico, y se pueden también utilizar nombres simbólicos, llamados *etiquetas*, para designar a las direcciones de memoria (con lo cual se libera al programador de la necesidad de conocer las direcciones absolutas de la memoria). Por ejemplo, los códigos de operación de un lenguaje ensamblador para el modelo de ordenador definido en el apartado anterior podrían ser:

código de máquina	código ensamblador
0000	PAR
0001	CAR
0010	ALM
0011	SUM
0100	RES
0101	MUL
0110	SAI
0111	SAC
1000	SAS
...	...

Para el primero de los dos ejemplos anteriores, un posible programa en ensamblador sería:

```

A      MEM  1
B      MEM  1
C      MEM  1
D      MEM  1
      ...
      ...
      CAR  B
      SUM  C
      MUL  A
      ALM  D
      ...
      ...
      PAR
      FIN

```

El programa traductor (el ensamblador) deberá tomar estos códigos como datos de entrada y generar los códigos binarios en lenguaje de máquina.

Las cuatro primeras líneas del programa que hemos escrito no representan instrucciones de máquina, sino "seudoinstrucciones". Se trata, en realidad, de instrucciones para el ensamblador. Concretamente, le dicen a éste que reserve cuatro palabras de la memoria y que las identifique mediante las etiquetas *A*, *B*, *C* y *D*. El ensamblador, al ejecutarse, se encargará de atribuir direcciones reales a esos nombres. Si, por ejemplo, atribuye a "*B*" la dirección 1, entonces la instrucción "CAR B" se traducirá por "0001000000000001", que es la misma que aparecía en el programa escrito en lenguaje de máquina.

Como puede observarse, gracias al uso de etiquetas el programador no tiene que preocuparse de direcciones numéricas reales de la memoria.

"FIN" es otra seudoinstrucción. Le indica al ensamblador el final del programa fuente.

Para el segundo ejemplo, podemos escribir este programa en ensamblador:

```

A      MEM  1
B      MEM  1
UNO    CEN  1
      SAS  LEER
      SAS  LEER
BUCLE  CAR  A
      RES  B
      SAC  FIN
      CAR  A
      SUM  UNO

```

```

                                ALM   A
                                SAI   BUCLE
FIN      PAR
LEER     (a partir de aquí vendrían las
          instrucciones del subprograma de
          lectura de datos)
          ...
          ...
          FIN

```

Hemos supuesto que el ensamblador reconoce la seudoinstrucción "CEN": crear una constante entera.

El modelo de ordenador que hemos utilizado para estos ejemplos es muy simple. La máquinas reales disponen generalmente de muchos registros y modos de direccionamiento que es preciso conocer para poder programarlas en lenguaje ensamblador.

2.2.3 Lenguajes macroensambladores

En los lenguajes *macroensambladores* hay *macroinstrucciones* (abreviadamente, "macros"), instrucciones que en el proceso de traducción (ensamblaje) se traducen no en una sino en varias instrucciones de máquina. Ello facilita algo la tarea de programación, pero sigue siendo válido lo dicho acerca de la necesidad de conocer los detalles de la estructura de la máquina.

Existen "macros del sistema", previstas en el propio lenguaje (especialmente para operaciones de comunicación con los periféricos), y, además, el programador puede definir sus propias "macros" para secuencias de instrucciones que tenga que repetir varias veces en su programa. La diferencia con los subprogramas es que el ensamblador *expande* las "macros", es decir, en el proceso de ensamblaje, cada vez que encuentra una genera toda la secuencia de instrucciones de máquina que le corresponde.

2.3 Lenguajes de alto nivel

2.3.1 Sentencias

Un programa en un lenguaje de alto nivel está compuesto por una sucesión de *sentencias* (normalmente, el nombre "instrucción" se reserva para las instrucciones del nivel de máquina). Aquí hay un pequeño problema terminológico que conviene aclarar para evitar confusiones. En efecto, en teoría de lenguajes llamamos "sentencia" a cualquier cadena perteneciente a un lenguaje (capítulo 2, apartado 3); de acuerdo con esto, una sentencia sería cualquier programa *completo* que fuera sintácticamente correcto. Sin embargo, cuando hablamos de lenguajes de programación solemos entender por "sentencia" no

un programa completo, sino cada una de las unidades elementales que forman parte de él y que tienen un significado en la definición del lenguaje. Así, en el lenguaje Pascal, existen sentencias de asignación como

```
x := 3
```

(sustituir el valor que tenga la variable *x* por la constante 3), sentencias de acción condicional, como

```
if EB then S1 else S2
```

(si el resultado de evaluar la expresión booleana *EB* es "true" (verdadero) entonces ejecutar la sentencia *S1*, si no, *S2*), sentencias de iteración, como

```
while EB do S
```

(ejecutar la sentencia *S* mientras el resultado de la expresión booleana *EB* sea "true"), etc.

El primero de los programas cuya codificación en lenguaje de máquina y en ensamblador vimos más arriba se podría codificar así en Pascal:

```
program aritmetico(input, output);  
begin  
  read(A,B,C);  
  D := A*(B+C);  
  write(D)  
end.
```

donde "begin" y "end" son "palabras clave" (símbolos terminales en la terminología de la teoría de lenguajes) que indican el principio y el final de un programa, y, en general, de un grupo cualquiera de sentencias. El programa consta de tres sentencias: una de lectura ("read"), otra de asignación y una final de escritura ("write"). El símbolo ";" separa cada sentencia de la siguiente.

El programa en Pascal para el segundo de los ejemplos podría ser:

```
program igualar (input,output);  
begin  
  read(A,B);  
  while A<>B do A := A+1  
end.
```

Otros ejemplos de programas en Pascal pueden encontrarse en los capítulos 3 y 7 del tema "Algoritmos".

2.3.2 Procedimientos y funciones

Una característica muy importante de todos los lenguajes de alto nivel es que facilitan el uso de *procedimientos* ("procedures"). Un procedimiento es la generalización de un subprograma: es una unidad programada independiente que se ejecuta cuando se la llama desde un programa, o desde otro procedimiento, o, incluso, desde el mismo procedimiento.

Un procedimiento consta de tres partes: su *nombre* (con el que se le llama), una lista de *parámetros formales* y el *cuerpo*, donde se incluyen las sentencias que forman el procedimiento. Por ejemplo, un procedimiento en Pascal para calcular el factorial de un número entero mediante una iteración es:

```
procedure factorial(n:integer; var fact:integer);
var i:integer;
begin
    fact := 1;
    for i := 2 to n do fact := fact * i
end;
```

El procedimiento utiliza dos parámetros formales: *n* es un parámetro de entrada, donde se da el número cuyo factorial hay que calcular; *fact* es un parámetro de salida, donde el procedimiento devuelve el valor calculado del factorial, y va precedido de la palabra clave "var" porque es una variable cuyo valor queda determinado tras la ejecución del procedimiento. Si, por ejemplo, desde un programa queremos calcular el factorial de 5, dejando el resultado en una variable de este programa llamada "f", incluiremos en él la sentencia de llamada

```
factorial(5,f);
```

"5" y "f" son los *parámetros reales*, que, cuando el procedimiento se ejecute atendiendo a esta llamada, irán a sustituir a los parámetros formales.

Un procedimiento puede comunicarse también con el programa que le llama (o sea, recibir datos o entregar resultados) mediante el uso de *variables globales*, es decir, variables definidas en el programa y accesibles también desde el procedimiento. (En el procedimiento "factorial" que hemos escrito, "i" es una *variable local*, porque está definida dentro del cuerpo del procedimiento).

Una *función* es otro tipo de subprograma. Al igual que un procedimiento, puede tener parámetros formales, y compartir variables globales, pero, a diferencia de él, devuelve directamente un valor al programa que la llama. Por ejemplo, para el factorial podemos escribir esta función en Pascal:

```

function factorial(n:integer):integer;
  var i,fact:integer;
  begin
    fact:=1;
    for i:=2 to n do fact:=fact*i;
    factorial:=fact
  end;

```

La llamada desde el programa para calcular el factorial de 5 y dejarlo en la variable *f* implica ahora el uso de una sentencia de asignación, para asignar a *f* el valor que devuelve la función:

```
f:=fact(5)
```

Como hemos dicho, los procedimientos y las funciones pueden utilizarse en la mayoría de los lenguajes, y, desde luego, en Pascal, de manera recursiva. Por ejemplo, otra forma de escribir una función para el cálculo del factorial puede ser:

```

function factorial(n:integer):integer;
  begin
    if n<2 then factorial:=1
    else factorial:=n*factorial(n-1)
  end;

```

donde, como vemos, la función "se llama" a sí misma.

El uso de procedimientos y funciones permite desarrollar programas de manera modular y jerárquica, materializando así uno de los principios básicos de la programación estructurada (tema "Algoritmos", capítulo 3, apartado 4).

2.3.3 Tipos de datos

Los lenguajes modernos tienden a ser "fuertemente tipados" (desafortunada traducción de "strongly typed"). Esto quiere decir que todas las variables que intervienen en un programa, en una función o en un procedimiento, tienen un *tipo*, y este tipo debe ser declarado explícitamente mediante las oportunas *sentencias de declaración*. Así, en los últimos ejemplos sobre procedimientos y funciones hemos incluido la declaración "integer" (entero) para las variables que intervenían, y también para la misma función, puesto que ésta devuelve un valor que también tiene un tipo (en este caso, entero). En general, un *tipo de datos* es un conjunto de valores posibles con unas operaciones asociadas. Aunque la obligación de declarar los tipos pueda, a primera vista, parecer un engorro innecesario (en relación con lenguajes clásicos, como FORTRAN, en los que no es preciso tal cosa) la ventaja es que la fiabilidad de los programas

mejora, pues gracias a eso se ponen de manifiesto, durante la compilación o la ejecución, muchos errores que de otro modo pasarían inadvertidos o serían muy difíciles de localizar.

Pascal incluye en su definición cuatro tipos "estándar": *integer* (el valor de la variable puede ser cualquier número entero comprendido en el rango de representación de la máquina), *boolean* (la variable sólo puede tomar dos valores, *true* y *false*), *real* (un número real comprendido en el rango de representación) y *char* (un carácter). A partir de ellos, el programador puede definir *tipos estructurados*: *array*, *record*, etc., cuyos componentes pueden ser también estructurados. Es posible, pues, construir *jerarquías de estructuras de datos*, insistiendo así en el diseño modular y jerárquico de los programas. Además, algunos compiladores para Pascal utilizan ampliaciones del lenguaje que permiten programar con orientación a objetos.

2.4 Lenguajes declarativos

2.4.1 Programación imperativa y programación declarativa

Los lenguajes de alto nivel comentados en el apartado anterior liberan al programador de las características concretas de la máquina en la que se ejecutan finalmente los programas. No obstante, lo mismo que los de bajo nivel, son *lenguajes imperativos*. Esto quiere decir que el programador, para resolver un problema, ha de diseñar un algoritmo y materializarlo en un programa que le diga a la máquina real (caso de los de bajo nivel) o a la máquina virtual (caso de los de alto nivel), paso a paso, qué tiene que hacer para resolver el problema. La programación orientada a objetos (cuyas ideas esenciales se han estudiado en el capítulo 4 del tema "Algoritmos") permite elevar el nivel de abstracción, pero no deja de ser una programación imperativa (desde el momento en que los "métodos" son programas imperativos).

La idea de los *lenguajes declarativos* es que el programador sólo tenga que declarar o especificar el problema, y que sea el procesador de lenguaje el que se encargue del algoritmo. Dicho de otro modo, que el programador indique "qué" es lo que hay que resolver y la máquina (virtual) se ocupe de los detalles del "cómo". En realidad, esta idea no es nueva, y está ya implícita en ciertos lenguajes diseñados para aplicaciones específicas, como cálculos estadísticos (*SPSS*, *BMDP*, etc.), consulta de bases de datos (*IMS*, *NATURAL*, etc.), simulación (*CSMP*, *GPSS*, etc.). Son también declarativos los lenguajes llamados *generadores*. Por ejemplo, un generador de informes (*RPG*: "report program generator") acepta descripciones de la estructura de representación de ciertos datos y del formato deseado de salida y genera un programa para imprimir los datos. Pero los dos "estilos" de programación declarativa más conocidos son los correspondientes a la programación funcional y a la programación lógica.

2.4.2 Programación funcional

El fundamento teórico de la programación funcional tiene su origen en el problema de la computabilidad de funciones matemáticas, y, concretamente, en el llamado "cálculo lambda". Sin entrar en esta base matemática, digamos simplemente que un programa funcional es un conjunto de ecuaciones que definen funciones a partir de otras funciones más sencillas o de primitivas. Por ejemplo, en el programa

```
max2[x,y] := if x > y then x else y
```

```
max3[x,y,z] := max2[max2[x,y],z]
```

la primera sentencia define la función "max2" a partir de las primitivas "if", ">", "then" y "else", y la segunda define la función "max3" a partir de "max2". Es importante señalar que el símbolo ":"=" no representa, como en algunos lenguajes imperativos, una asignación; aquí significa "se define como". De hecho, en LISP, por ejemplo, no se utiliza este símbolo, sino la palabra clave "defun".

Las definiciones de funciones pueden ser recursivas. Por ejemplo:

```
fact[N] := if N = 0 then 1 else N*fact[N-1]
```

Un concepto fundamental en programación funcional es el de la *transparencia referencial*. Una función es referencialmente transparente, o *pura*, si su efecto queda definido exclusivamente por el valor dado a sus parámetros. Para ello es preciso que la función no tenga *efectos laterales*, es decir, que no altere ni el valor de sus parámetros ni variables globales. La idea básica de la programación funcional es la de expresar todos los algoritmos mediante aplicaciones de funciones puras (por ello, a los lenguajes funcionales se les llama también *lenguajes aplicativos*).

En principio, también puede hacerse programación funcional con lenguajes imperativos. Basta para ello utilizar funciones que no manejen variables globales ni alteren sus parámetros, como las dos funciones escritas en Pascal en el apartado 2.3.2. El problema está en que el único modo de obtener un resultado es a través del valor del objeto que devuelve la función, y en los lenguajes imperativos clásicos el tipo de este objeto está muy limitado.

En un lenguaje funcional los objetos pueden ser *listas*. Una lista es una secuencia finita de elementos, donde cada elemento es un "átomo" (numérico, como "109" o simbólico, como "ÁTOMO") u otra lista. Las listas permiten una representación uniforme y versátil de gran variedad de objetos, aunque a veces resulten de difícil lectura. Por ejemplo, las expresiones algebraicas

$$\begin{array}{l} x + y \\ \text{y} \\ ax^2 + bx + c = 0 \end{array}$$

se escribirían así con notación de lista:

$$\begin{array}{l} (\text{suma } x \ y) \\ \text{y} \\ (\text{igual}(\text{suma}(\text{mul } a \ (\text{cuad } x))(\text{suma}(\text{mul } b \ x) \ c)) \ 0) \end{array}$$

Para manejar listas, los lenguajes funcionales suelen tener incorporada una *función de construcción* que permite encadenar dos listas. Esta función se puede escribir de forma prefija: `cons [L1, L2]` o de forma infija: `L1 . L2`. Por ejemplo, la lista (a, b, c) se puede construir así a partir de los átomos a , b y c :

`cons (a, cons (b, (cons (c, ())))`

o bien:

`(a . (b . (c . ())))`

donde "`()`" representa una lista especial: la *lista vacía*.

La facilidad de manejo de expresiones simbólicas y de jerarquías de objetos representadas como listas es la razón por la que LISP, el lenguaje funcional más antiguo, es también el más utilizado en inteligencia artificial, donde el procesamiento simbólico predomina sobre el numérico. Hay muchos otros lenguajes funcionales más modernos, pero menos conocidos que LISP, como CAML, LML, Miranda, Hope, Haskell...

2.4.3 Programación lógica

El fundamento teórico de la programación lógica es el cálculo de predicados de primer orden. El lenguaje más conocido es Prolog, en el que las sentencias son cláusulas de Horn (tema "Lógica", capítulo 4, apartado 4.5) compuestas con fórmulas atómicas del cálculo de predicados y escritas de un modo especial: primero se escribe la *cabeza* de la cláusula (o consecuente del condicional), si existe, y luego, separado por el símbolo "`:-`" (o "`←`"), el *cuerpo* (o antecedente). Así, la *regla*

$A \text{ :- } B, C, D.$

equivale a la sentencia de la lógica

$B \wedge C \wedge D \rightarrow A$

donde A , B , C y D representan fórmulas atómicas.

Por ejemplo, un programa que declara unos hechos de parentesco y define la relación "abuelo" es éste, formado con dos hechos y dos reglas:

```
madre(ana, luis) .
padre(josé, ana) .
abuelo(X, Z) :- padre(X, Y), padre(Y, Z) .
abuelo(X, Z) :- padre(X, Y), madre(Y, Z) .
```

Este programa corresponde exactamente al ejemplo 1.7.4 visto en lógica de predicados (tema "Lógica", capítulo 4), salvo que, como puede observarse, la definición de la relación "abuelo" se ha descompuesto en dos sentencias para que sean cláusulas de Horn. Un convenio sintáctico de Prolog es que los nombres de las variables empiezan con letra mayúscula, mientras que los nombres de las constantes (y también los de los predicados) empiezan con minúscula.

Si se dispone de un intérprete de Prolog y del anterior programa en memoria, se pueden hacer consultas. Por ejemplo:

```
?- abuelo(josé, luis) .
SI

?- abuelo(josé, ana) .
NO

?- abuelo(X, luis) .
X = josé
```

Para obtener las respuestas, el intérprete de Prolog aplica resolución y refutación (tema "Lógica", capítulo 4, apartado 4.7; véase también, más adelante, el apartado 5.4).

En Prolog se pueden definir también relaciones de modo recursivo. Así, un programa para definir "antepasado" (tema "Lógica", capítulo 4, ejemplo 1.7.5) es:

```
pad_o_mad(X, Y) :- padre(X, Y) .
pad_o_mad(X, Y) :- madre(X, Y) .
antepasado(X, Y) :- pad_o_mad(X, Y) .
antepasado(X, Y) :- pad_o_mad(X, Z), antepasado(Z, Y) .
```

2.5 Entornos de desarrollo

Asociadas a un lenguaje, pueden diseñarse "herramientas", que son programas para facilitar su uso, y, en general, mejorar la productividad del desarrollo.

llo de programas. El conjunto integrado de herramientas asociado a un lenguaje se llama *entorno de programación*, y, dependiendo del número de herramientas disponibles y de las facilidades que ofrecen, puede ser más o menos potente. Ya hemos hablado de un tipo de herramienta: los procesadores de lenguaje (traductores e intérpretes). Los *editores* permiten escribir y modificar con facilidad el programa fuente, los *depuradores* ayudan a detectar y corregir errores, los *enlazadores* (o *montadores*) permiten obtener un programa binario ejecutable a partir del resultado de la compilación de varios módulos, etc. Otras herramientas más avanzadas ayudan a la especificación de programas y permiten demostrar, antes de su ejecución, si son correctos (satisfacen las especificaciones) o no. Típicamente, un entorno de programación incluye varias de estas herramientas y facilita su uso. Por ejemplo, si el programador somete un programa al compilador y éste encuentra un error, el entorno hace que automáticamente pase a ejecutarse el editor para que el programador pueda corregirlo.

Las herramientas integradas pueden no estar orientadas exclusivamente a un solo lenguaje, sino, de manera más general, al desarrollo de aplicaciones. Se habla entonces de *entornos de desarrollo*. Un ejemplo son los entornos para el desarrollo de sistemas basados en conocimiento, integrados por herramientas como las que hemos comentado en el apartado 1.4 del capítulo 6 del tema "Lógica".

Para el diseño sistemático y riguroso de todas estas herramientas es necesario disponer de una *definición formal* del lenguaje tratado por la herramienta. Veamos los fundamentos de estas definiciones formales.

3. Definiciones sintácticas

3.1 Notación de Backus

La sintaxis de la práctica totalidad de los lenguajes de programación puede expresarse mediante una gramática libre de contexto (salvo algunas excepciones que comentaremos en el apartado 3.2). La notación de Backus, generalmente conocida como "*BNF*" (de "Backus Normal Form", o "Backus-Naur Form") no es más que una forma de escribir las producciones, que sólo difiere de la que hemos visto en el capítulo 2 en que en lugar de " \rightarrow " se utiliza " $::=$ ", y, además, se puede abreviar la escritura de varias reglas que compartan el mismo antecedente:

$$\begin{aligned} A &::= \alpha_1 \\ A &::= \alpha_2 \\ A &::= \alpha_3, \\ &\dots \end{aligned}$$

en una sola línea, de este modo:

$$A ::= \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots$$

Normalmente, para mayor claridad, no se utilizan letras aisladas para los símbolos, sino palabras completas. Para distinguir a los símbolos del alfabeto auxiliar de los del alfabeto terminal, los primeros se escriben, como ya hemos hecho en algunos ejemplos anteriores, encerrados entre paréntesis angulares: $\langle \text{sentencia} \rangle$, $\langle \text{identificador} \rangle$, etc., y a los del terminal sin paréntesis: *begin*, *if*, etc. Concretamente, el símbolo inicial de una gramática para un lenguaje de programación puede ser $\langle \text{programa} \rangle$. Todo programa correcto, $x \in L$, podrá derivarse en esa gramática mediante aplicación primero de la regla que tenga $\langle \text{programa} \rangle$ como antecedente y después de otras reglas en el orden adecuado para obtener tal programa.

Hay, además, algunas extensiones de la notación *BNF* original muy útiles para abreviar la escritura de las reglas recursivas. Los corchetes indican que lo que figura dentro puede repetirse cualquier número de veces (o no aparecer), y los paréntesis cuadrados lo mismo, pero apareciendo al menos una vez. Así, en la gramática para expresiones aritméticas con números positivos y operaciones "+" y "*" definida en el ejemplo 5.5 del capítulo 3, las dos primeras reglas:

$$\langle \text{EA} \rangle ::= \langle \text{EA} \rangle + \langle \text{TERM} \rangle$$

$$\langle \text{EA} \rangle ::= \langle \text{TERM} \rangle$$

(una expresión es o bien un solo término o bien una suma de términos) pueden escribirse en BNF así:

$$\langle \text{EA} \rangle ::= \langle \text{TERM} \rangle \{ + \langle \text{TERM} \rangle \}$$

Y las que generan $\langle \text{CTE} \rangle$:

$$\langle \text{CTE} \rangle ::= \langle \text{CTE} \rangle \langle \text{DIG} \rangle$$

$$\langle \text{CTE} \rangle ::= \langle \text{DIG} \rangle$$

(una constante es una secuencia de uno o más dígitos) así:

$$\langle \text{CTE} \rangle ::= \langle \text{DIG} \rangle \{ \langle \text{DIG} \rangle \}$$

o bien:

$$\langle \text{CTE} \rangle ::= [\langle \text{DIG} \rangle]$$

3.2 Gramáticas para lenguajes de programación

Decíamos más arriba que la sintaxis de los lenguajes de programación, salvo excepciones, puede expresarse mediante gramáticas libres de contexto. Estas excepciones se refieren a algunos aspectos de la definición de los lenguajes. Por poner un ejemplo, en un lenguaje como Pascal es obligatorio que se declare el tipo de las variables previamente a su uso en el programa (apartado 2.3.3). Formalmente, esto se traduce en que dada una cadena (programa) de la forma

$$\alpha \text{ begin } \beta x \gamma \text{ end.}$$

donde x es una variable y α, β, γ representan el resto del programa, si x figura en α (y suponiendo que α, β, γ están correctamente construidas: por ejemplo, x en α debe ir precedida de "var") entonces la cadena pertenece al lenguaje, y si no, no. Pues bien, se puede demostrar que un lenguaje así no es libre de contexto.

Lo que se suele hacer es ignorar estas excepciones y definir (de manera algo forzada) el lenguaje mediante una gramática libre de contexto. Por ejemplo, en los compiladores la detección de variables no declaradas no tiene lugar en la fase de "análisis sintáctico" (en la que se aplican las reglas libres de contexto), sino después, en la de "análisis semántico" (apartado 5.2.1).

Por otra parte, dentro del tipo general de gramáticas libres de contexto existen clases definidas por ciertas restricciones sobre la forma de las reglas. Por ejemplo, en el apartado 6 del capítulo 14 hemos mencionado un subconjunto llamado "gramáticas deterministas". Los reconocedores, en la práctica, van a materializarse en algoritmos que realizan un análisis sintáctico del programa fuente, y estos algoritmos pueden ser más sencillos y más eficaces si la gramática cumple con ciertas restricciones. De hecho, cuando se diseña un lenguaje nuevo las decisiones sobre la gramática del mismo vienen muy condicionadas por los algoritmos de reconocimiento que luego habrán de programarse.

En los ejemplos que desarrollaremos a continuación vamos a imponer dos condiciones que conducen a un tipo de gramática determinista y a un algoritmo de reconocimiento sencillo, como veremos en el apartado 5.2.3.

La primera condición es que si existe una regla de la forma

$$A ::= \alpha | \beta$$

entonces no puedan derivarse a partir de α y β cadenas que comiencen con un mismo símbolo terminal. En particular, no sería lícita una regla como

$$A ::= \alpha\beta_1 | \alpha\beta_2$$

En éste, como en otros muchos casos, es fácil encontrar otras reglas que generen las mismas cadenas y que cumplan la condición enunciada. Concretamente, basta con introducir un nuevo símbolo auxiliar, R , y sustituir la regla anterior por:

$$A ::= \alpha R$$

$$R ::= \beta_1 \mid \beta_2$$

Ese procedimiento de "sacar factor común" puede conducir a algo que normalmente está prohibido en la definición de las gramáticas libres de contexto, pero que pese a todo se utiliza para cumplir la condición. Se trata de que, excepcionalmente, el consecuente de una regla puede ser la cadena vacía. Por ejemplo, en el caso de la regla

$$A ::= \alpha \mid \alpha\beta$$

que sustituiremos por

$$A ::= \alpha R$$

$$R ::= \beta \mid \lambda$$

(Obsérvese que, pese a tener la regla $R ::= \lambda$, las cadenas derivadas a partir de A siguen siendo de longitud no decreciente).

Este caso aparece, por ejemplo, en la sentencia "if" de los lenguajes de programación, con sus dos posibilidades:

$$\begin{aligned} \langle \text{sentencia} \rangle &::= \text{if } \langle \text{expresión} \rangle \text{ then } \langle \text{sentencia} \rangle \mid \\ &\quad \text{if } \langle \text{expresión} \rangle \text{ then } \langle \text{sentencia} \rangle \text{ else } \langle \text{sentencia} \rangle \end{aligned}$$

La segunda condición es que la gramática no sea *recursiva por la izquierda*, lo que quiere decir que no puedan darse derivaciones de la forma $A \xRightarrow{*} A\alpha$. Existe un algoritmo general para transformar la recursión por la izquierda en recursión por la derecha. Veamos aquí el caso más sencillo y más frecuente, que es el de la existencia de dos reglas como:

$$A ::= \alpha \mid A\beta$$

que pueden sustituirse por estas otras:

$$A ::= \alpha R$$

$$R ::= \beta R \mid \lambda$$

o bien, usando la notación *BNF* ampliada:

$$A ::= \alpha \{ \beta \}$$

De hecho, esta transformación es la que hemos utilizado al final del apartado anterior cuando hemos reescrito en notación *BNF* algunas reglas de una gramática para expresiones aritméticas.

3.3 Gramática para un lenguaje ensamblador

Como primer ejemplo de uso de la notación *BNF* para la definición formal de un lenguaje de programación, vamos a considerar un lenguaje ensamblador muy sencillo: el mismo cuya sintaxis y semántica hemos definido informalmente en el apartado 2.2.2.

El inconveniente de las definiciones informales es que suelen ser incompletas y ambiguas. Por ejemplo, podemos decir que un programa en nuestro lenguaje ensamblador es una secuencia de instrucciones terminada por la seudoinstrucción "FIN". Esto podría conducirnos a escribir así la primera regla:

$$\langle \text{programa} \rangle ::= \langle \text{instrucción} \rangle \langle \text{programa} \rangle \text{FIN}$$

Ahora bien, esta regla indicaría que las instrucciones deberían ir una tras otra en la misma línea, mientras que, como hemos visto en los ejemplos del apartado 2.2.2, se escriben cada una en una línea separada. Para reflejar esta imposición en la forma de la regla, introduciremos un símbolo terminal especial, "cl" (cambio de línea) entre $\langle \text{instrucción} \rangle$ y $\langle \text{programa} \rangle$. Por otra parte, la definición anterior de programa como secuencia de instrucciones es incompleta: puede haber también seudoinstrucciones (MEM y CEN). Además, la regla anterior tiene un problema: la última línea de un programa sería el símbolo terminal "FIN" con la letra "F" en la primera columna de la línea. Pero, como se desprende de los ejemplos, las etiquetas se escriben justamente desde la primera columna, por lo que el ensamblador no sabría que "FIN" es una seudoinstrucción y no una etiqueta. La regla sintáctica es que los códigos de las instrucciones y seudoinstrucciones no pueden empezar en la primera columna: tienen que llevar delante al menos un espacio en blanco. Llamaremos $\langle \text{separ} \rangle$ a la categoría sintáctica que significa "uno o más espacios en blanco". Finalmente, vamos a considerar la posibilidad de escribir comentarios. Un comentario es un texto que el procesador (ensamblador, en este caso) debe ignorar. Supondremos que los comentarios se escriben en líneas diferentes de las que corresponden a las instrucciones y seudoinstrucciones. De acuerdo con todo esto, escribimos dos reglas:

$$(1) \langle \text{programa} \rangle ::= \langle \text{línea} \rangle \text{cl} \langle \text{programa} \rangle \langle \text{separ} \rangle \text{FIN}$$

$$(2) \langle \text{línea} \rangle ::= \langle \text{instrucc} \rangle \langle \text{seudoinstrucc} \rangle \langle \text{coment} \rangle$$

Tenemos ahora que introducir nuevas reglas para obtener derivaciones a partir de <instrucc>, <seudoinstrucc> y <coment>. Sabemos que una instrucción (y una seudoinstrucción) puede llevar etiqueta o no. La etiqueta se escribe a partir de la primera columna de la línea. Supondremos que el código de operación no tiene por qué escribirse en columnas determinadas: basta con que esté separado del final de la etiqueta por uno o más espacios en blanco ("<separ>"), y si no hay etiqueta debe ir precedido por <separ>. Hay, además, ciertas instrucciones que tienen que ir acompañadas de una dirección de memoria (SUM, RES, etc.); las seudoinstrucciones CEN y MEM tienen que ir seguidas de un número. Se separará esta dirección o este número del código de operación mediante <separ>. Otras instrucciones, como PAR, CMP (complementar el acumulador), etc., no llevan nada más. En cuanto a los comentarios, supondremos que las líneas correspondientes se distinguen porque en la primera columna tienen el carácter "*" y luego cualquier texto (cadena de caracteres). Todo ello se puede formalizar con las siguientes reglas:

- (3) <instr> ::= <eti><separ><resto-instr>|<separ><resto-instr>
- (4) <resto-instr> ::= <oper-con-dir><separ><dir>|<oper-sin-dir>
- (5) <seudoinstr> ::= <eti><separ><resto-seudo>|<separ><resto-seudo>
- (6) <resto-seudo> ::= <cod-seudo><separ><num>
- (7) <coment> ::= *{<carácter>}

Tenemos ahora nuevas categorías sintácticas para las que hemos de introducir nuevas reglas: <eti>, <dir>, etc. Escribamos ya, sin extendernos en más explicaciones, el resto de las reglas:

- (8) <separ> ::= bl{|bl} ("bl" indica "espacio en blanco")
- (9) <letra-may> ::= A|B|C|...|Z
- (10) <letra-min> ::= a|b|c|...|z
- (11) <dígito> ::= 0|1|2|3|4|5|6|7|8|9
- (12) <carác-esp> ::= *|?|/|bl|...
- (13) <carácter> ::= <letra-may>|<letra-min>|<dígito>|<carác-esp>
- (14) <eti> ::= <letra-may>{<letra-may>|<dígito>}
- (15) <num> ::= <dígito>|{<dígito>} (comparar con el ejemplo 4.4 del capítulo 2)

(16) $\langle \text{dir} \rangle ::= \langle \text{eti} \rangle | \langle \text{num} \rangle$

(17) $\langle \text{oper-con-dir} \rangle ::= \text{CAR} | \text{ALM} | \text{SUM} | \dots$

(18) $\langle \text{oper-sin-dir} \rangle ::= \text{PAR} | \text{CMP} | \dots$

(19) $\langle \text{cod-seudo} \rangle ::= \text{MEM} | \text{CEN}$

Al hilo de esta definición formal, nos parece interesante hacer algunas observaciones:

a) La gramática definida por las anteriores reglas es del tipo libre de contexto, porque en todas ellas el antecedente es un símbolo del alfabeto auxiliar; además, cumple las restricciones enunciadas en el apartado 3.2.

b) La formalización obliga a dejar perfectamente claros todos los detalles, que en una descripción informal pueden olvidarse. Por ejemplo, una etiqueta, según la regla 14, tiene que estar formada por letras mayúsculas y dígitos, empezando por una letra, cosa que no habíamos dicho en nuestra presentación informal. Esto, ya de por sí, es una ventaja. Otra no menos importante es que la definición formal permite diseñar de modo sistemático los procesadores de lenguaje.

c) Según la regla 1, un programa puede tener un número ilimitado de líneas. De igual modo, las etiquetas (regla 14) y los números (regla 15) no tienen limitación de caracteres. Evidentemente, en la práctica (en la "implementación") han de introducirse restricciones.

d) Las constantes de la seudoinstrucción CEN (reglas 19, 6 y 15) son siempre números positivos. Es fácil ampliar la gramática para que se admitan también los negativos. Por ejemplo, con una nueva regla:

$$\langle \text{cte} \rangle ::= \langle \text{num} \rangle | + \langle \text{num} \rangle | - \langle \text{num} \rangle$$

y las oportunas modificaciones para que en la generación de CEN (no así en la de MEM) aparezca $\langle \text{cte} \rangle$.

e) Existen otras muchas posibilidades de definición de reglas y categorías sintácticas (símbolos auxiliares) para el mismo lenguaje, que corresponden a gramáticas equivalentes (capítulo 2, apartado 3).

3.4 Gramática para un lenguaje de alto nivel

Tomaremos como ejemplo sencillo de un lenguaje de alto nivel un pequeño subconjunto de Pascal. Presentaremos primero de manera informal su sintaxis y su semántica.

Para simplificar, sólo consideraremos un tipo de dato: entero ("integer"). En el programa pueden figurar constantes literales (por ejemplo, -3, 100, etc.) y constantes y variables denominadas mediante identificadores; estas últimas deben declararse al principio mediante sentencias de declaración. Por ejemplo:

```
program ejemplo;
  const menosdos = -2;
  diez = 10;
  var   i,j : integer;

begin

  ... {sentencias del programa}

end.
```

Las constantes y variables pueden combinarse con los operadores aritméticos "+", "-", "*" y "/" formando expresiones aritméticas, y con los operadores relacionales "=", "<", ">", "<=", ">=", "<>" formando condiciones que se evalúan como verdaderas (por ejemplo, "3>2") o falsas (por ejemplo, "3<2").

Las sentencias posibles son:

- La sentencia de asignación:
Sintaxis: `v := expresión`
Semántica: asocia a la variable `v` el valor resultante de evaluar la expresión (sustituyéndolo por el que tenía asociado anteriormente).
- La sentencia de acción condicional:
Sintaxis: `if condición then sentencia`
o bien:
`if condición then sentencia1 else sentencia2`
Semántica: si la condición se evalúa como verdadera, entonces se ejecuta la "sentencia1"; si se evalúa como falsa no se hace nada (primer caso) o se ejecuta la "sentencia2" (segundo caso).
- La sentencia de iteración:
Sintaxis: `while condición do sentencia`
Semántica: se ejecuta la sentencia repetidamente siempre que la evaluación de la condición sea verdadera.

- La sentencia de entrada de datos⁵:
 Sintaxis: `read(v)`
 Semántica: lee un valor del periférico de entrada y lo asigna a la variable `v`.
- La sentencia de salida de resultados:
 Sintaxis: `write(expresión)`
 Semántica: evalúa la expresión y escribe el resultado por el periférico de salida.

En todos los casos anteriores, "sentencia" puede ser una sentencia simple o compuesta por una secuencia de varias; en este caso deben ir separadas por ";", precedidas por "begin" y terminadas por "end". Los espacios en blanco y los cambios de línea no tienen significado, y pueden introducirse arbitrariamente para mejorar la legibilidad.

Una definición formal de la sintaxis del lenguaje es la dada por las siguientes reglas en notación BNF:

- (1) `<programa> ::= program <identif>; <bloque>.`
- (2) `<bloque> ::= [const <decl-const>]0! [var <decl-var>]0! begin<sentencia>{;<sentencia>}end`
- (3) `<decl-const> ::= [<identif> = <cte>;]`
- (4) `<decl-var> ::= [<identif>{,<identif>:<tipo>;}]`
- (5) `<identif> ::= <letra>{<letra>|<dígito>}`
- (6) `<cte> ::= <num>|+<num>|-<num>`
- (7) `<num> ::= [<dígito>]`
- (8) `<letra> ::= a|b|...|z|A|B|...|Z`
- (9) `<dígito> ::= 0|1|...|9`
- (10) `<tipo> ::= integer`
- (11) `<sentencia> ::= begin<sentencia>{;<sentencia>}end|<identif>
 := <expresión>|
 if <condición> then <sentencia> <resto>|`

⁵ "read" y "write" no son, realmente, "sentencias": véase la observación al final de este apartado.

```
while <condición> do <sentencia>|
read(<identif>)| write(<expresión>)
```

(12) <resto> ::= else <sentencia>| λ

(13) <condición> ::= <expresión><oper><expresión>

(14) <oper> ::= = | < | > | <= | >= | <>

(15) <expresión> ::= <expr-s-sg>|+<expr-s-sg>|-<expr-s-sg>

(16) <expr-s-sg> ::= <término>|{+<término>|-<término>}

(17) <término> ::= <factor>|{*<factor>|/<factor>}

(18) <factor> ::= <cte>|<identif>|(<expresión>)

También aquí debemos señalar algunas observaciones:

a) La notación $[...]_0^1$ significa "una vez o ninguna".

b) Tal como se ha definido "<expresión>", los operadores "*" y "/" y tienen prioridad sobre "+" y "-", a menos que se anule tal prioridad con paréntesis (última parte de la regla 18). El procedimiento seguido para evitar la ambigüedad es el mismo que en el ejemplo 5.5 del capítulo 3.

c) Los símbolos "begin" y "end" cumplen una función con respecto a las sentencias similar a la que tienen los paréntesis con respecto a las expresiones. Si no existieran, una sentencia como

```
while C do S1;S2
```

sería ambigua, puesto que admitiría las dos derivaciones de la figura 5.3. Con nuestra gramática la única derivación posible es la primera, porque para la segunda habría que escribir:

```
while C do begin S1;S2 end
```

d) Sin embargo, nuestra gramática es ambigua en la parte que genera las sentencias "if". Por ejemplo, para la sentencia

```
if C1 then if C2 then S1 else S2
```

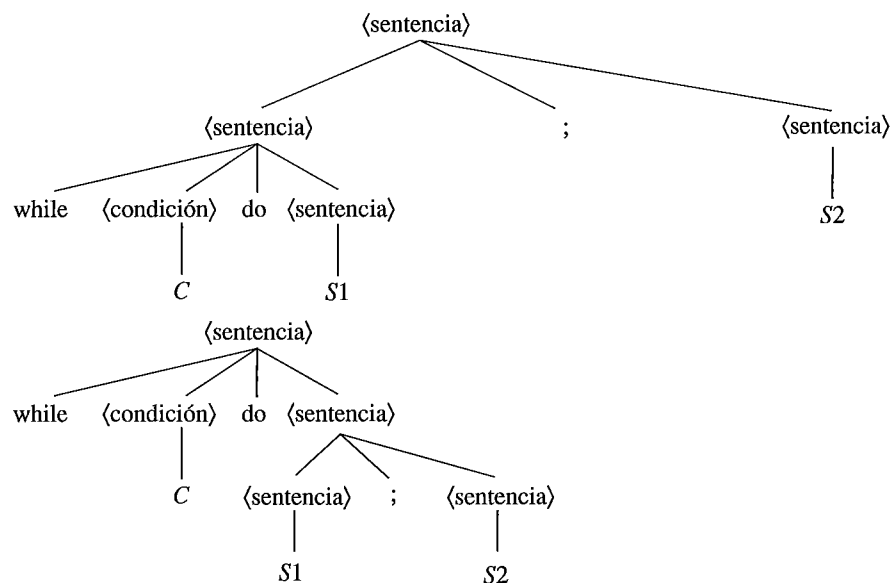


Figura 5.3.

tenemos los dos árboles de derivación de la figura 5.4. Normalmente, en los lenguajes con este tipo de construcciones se prefiere la interpretación correspondiente al primero de los árboles. En general, se sigue el criterio de emparejar cada "else" con el "then" inmediatamente anterior que no esté emparejado. El lector puede comprobar que si las reglas (11) y (12) se sustituyen por estas otras:

$\langle \text{sentencia} \rangle ::= \langle \text{s-emparejada} \rangle | \langle \text{s-no-emparejada} \rangle$

$\langle \text{s-emparejada} \rangle ::= \text{begin } \langle \text{sentencia} \rangle \{ ; \langle \text{sentencia} \rangle \} \text{ endl} \langle \text{identif} \rangle$
 $\quad \quad \quad := \langle \text{expresión} \rangle \text{ if } \langle \text{condición} \rangle \text{ then } \langle \text{s-emparejada} \rangle$
 $\quad \quad \quad \text{else } \langle \text{s-emparejada} \rangle |$
 $\quad \quad \quad \text{while } \langle \text{condición} \rangle \text{ do } \langle \text{sentencia} \rangle |$
 $\quad \quad \quad \text{read}(\langle \text{identif} \rangle) | \text{write}(\langle \text{expresión} \rangle)$

$\langle \text{s-no-emparejada} \rangle ::= \text{if } \langle \text{condición} \rangle \text{ then } \langle \text{resto} \rangle$

$\langle \text{resto} \rangle ::= \langle \text{sentencia} \rangle | \langle \text{s-no-emparejada} \rangle \text{ else } \langle \text{s-emparejada} \rangle$

se obtiene una gramática equivalente y no ambigua en la que las derivaciones para los "if anidados" siguen el criterio anterior.

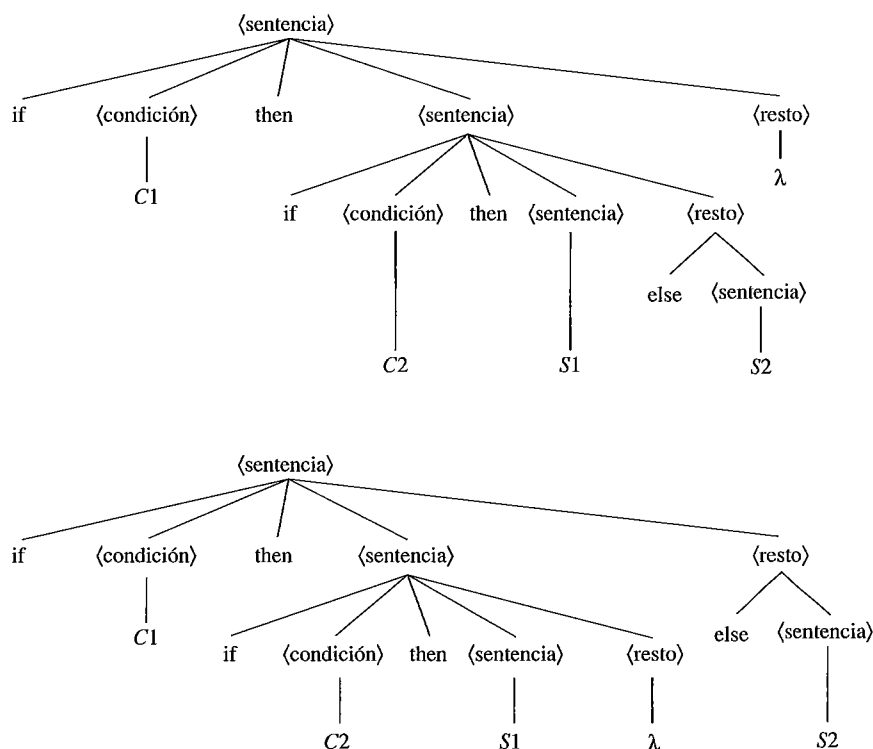


Figura 5.4.

e) Si el lector consulta en algún libro sobre Pascal su definición sintáctica, observará que no aparecen las sentencias "read" y "write". La explicación es que, en realidad, no son sentencias, sino llamadas a procedimientos o "rutinas de biblioteca" que forman parte de lo que se llama el "entorno de ejecución". Aquí las hemos incluido como sentencias porque para simplificar hemos prescindido de los procedimientos.

3.5 Diagramas sintácticos

Los diagramas sintácticos, o "diagramas de trenes", son construcciones gráficas que permiten presentar la misma información sintáctica que la notación BNF pero de una manera que resulta más fácil de interpretar para las personas. Un diagrama sintáctico es un grafo orientado con dos tipos de nodos: unos corresponden a los símbolos auxiliares, y se dibujan como cuadrados o rectángulos, y otros a los símbolos terminales, y se dibujan como círculos o rectán-

gulos con ángulos redondeados. Dado un conjunto de reglas BNF, el diagrama sintáctico correspondiente se puede construir siguiendo las siguientes normas:

a) A una producción del tipo

$$A ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

le corresponde el diagrama de la figura 5.5(a), donde cada rectángulo se sustituye según las normas que siguen.

b) Si α_i tiene la forma $\beta_1 \beta_2 \dots \beta_n$, se sustituye por el diagrama de la figura 5.5(b), en el que cada rectángulo se sustituye de acuerdo con las otras normas.

c) Si α_i (o β_i) tiene la forma $\alpha\{\beta\}$, se sustituye por la figura 5.5(c).

d) Si tiene la forma $\alpha\{\alpha\}$ (o, lo que es lo mismo, $[\alpha]$), se sustituye según indica la figura 5.5(d).

e) En el caso de $\alpha\{\beta\alpha\}$, tendríamos la figura 5.5(e)

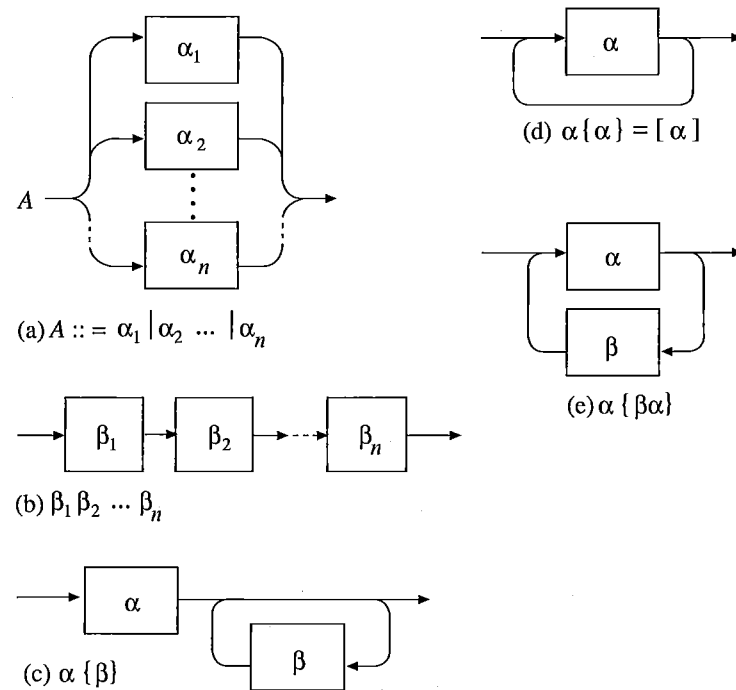


Figura 5.5.

El diagrama completo se presenta como un conjunto de subdiagramas en cada uno de los cuales pueden figurar símbolos auxiliares que se desarrollan en otros subdiagramas. La recursividad se manifiesta en el hecho de que el subdiagrama correspondiente a un símbolo auxiliar puede contener el nodo correspondiente a ese mismo símbolo, u otro cuyo subdiagrama lo contiene. Como ejemplo, en la figura 5.6 reproducimos el diagrama sintáctico correspondiente al lenguaje definido en el apartado 3.4.

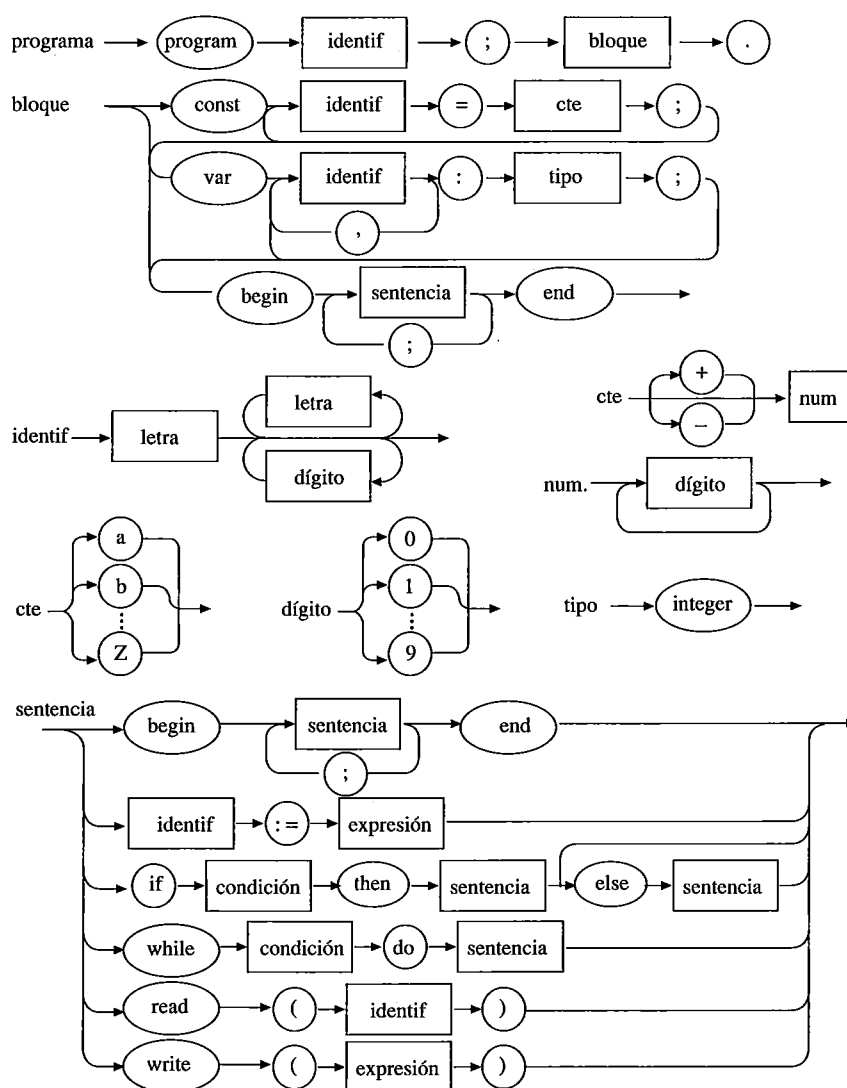


Figura 5.6.

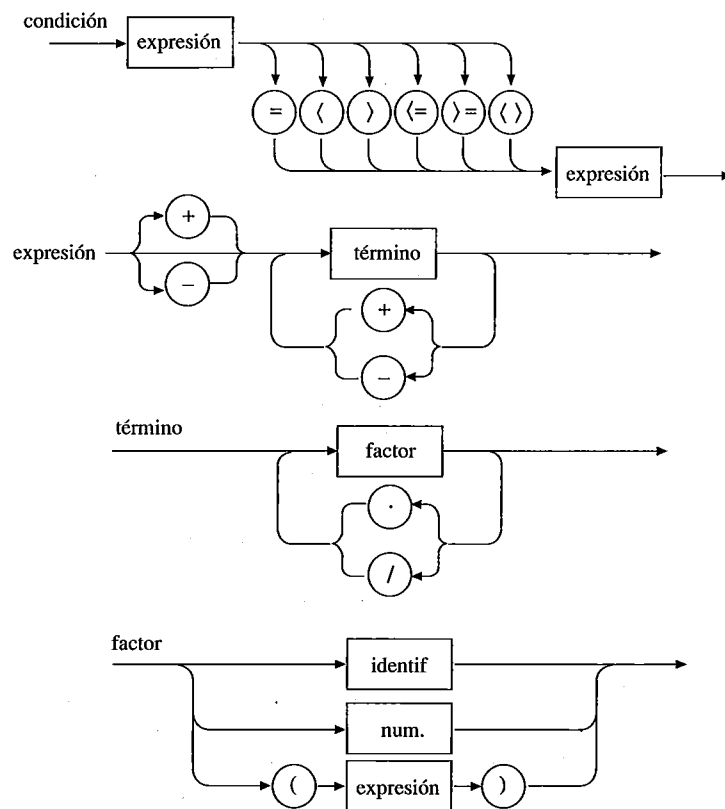


Figura 5.6. (cont.)

4. Definiciones semánticas

4.1 Objetivos

La definición sintáctica de un lenguaje permite comprobar si un determinado programa pertenece o no al lenguaje, y, como veremos en el apartado 5.2, reconstruir el árbol de derivación de ese programa. Pero no nos dice nada sobre el efecto de la ejecución del programa. Esto pertenece al campo de la semántica. Hemos visto en los anteriores ejemplos que una primera ventaja de la presentación formal de la sintaxis de un lenguaje procede del rigor y la con-

cisión del lenguaje matemático. El primer objetivo de la formalización de la semántica es, asimismo, definir el lenguaje sin la ambigüedad y la pérdida de detalles con que se hace cuando, como es habitual, se utiliza el lenguaje natural para ello. Por ejemplo, en la referencia básica del lenguaje Pascal⁶ puede leerse con relación a la sentencia de asignación: "especifica que un valor a computar se asigne a una variable. El valor está especificado por una expresión" (pág. 28). Pero esta definición es incompleta: ¿qué ocurre si la expresión no puede evaluarse, o si los tipos de la variable y del resultado de evaluar la expresión son distintos? Desde luego, en el libro citado se indican, también informalmente, las distintas posibilidades de compatibilidad entre tipos (pag. 33), pero no cabe duda de que lo mejor sería expresar escueta, clara y completamente el significado de la sentencia.

Es obvio que no sólo es importante que un programa sea sintácticamente correcto; es preciso que haga justamente lo que se pretende de él. Típicamente, en el proceso de desarrollo de un programa, una vez depurados los errores sintácticos se le somete a una serie de pruebas para examinar su comportamiento y se va refinando hasta que se está "razonablemente" seguro⁷ de que funciona correctamente. Este es un procedimiento rudimentario, costoso e inseguro. Una alternativa es desarrollar y aplicar métodos de prueba formal de programas. Es decir, demostrar, matemáticamente, que un programa dado hace lo que se supone que tiene que hacer. O, inversamente, sabiendo lo que tiene que hacer, demostrar constructivamente el programa. Este sería el segundo objetivo de la definición formal de la semántica: proporcionar una base rigurosa para razonar sobre los programas, y, por tanto, poder diseñar programas que, por el propio método de diseño, sean correctos.⁸

Un tercer objetivo está en relación con las ayudas informáticas al desarrollo de programas y los entornos de programación. La formalización de la sintaxis ha permitido diseñar herramientas para mejorar la productividad del desarrollo de software. Con la formalización de la semántica podrían conseguirse herramientas mucho más potentes. Por ejemplo, un método de diseño de programas es el basado en transformaciones: a partir de una especificación formal de lo que el programa debe hacer (escrita en un lenguaje de muy alto nivel), mediante transformaciones sucesivas con la ayuda de unas herramientas que elijan las reglas de transformación adecuadas, llegar al código de máquina final. Es como una generalización del proceso de compilación que veremos en el siguiente apartado. Pero al ser mucho más complejo, también resulta mucho más difícil diseñar las herramientas de modo que la semántica

⁶ El manual que describe la norma ISO: Jensen y Wirth (1985).

⁷ Recuérdese la "ley de Gilb": tema "Lógica", capítulo 2, apartado 1.5.

⁸ Hablar de "programa correcto" es común en el lenguaje cotidiano de la informática, y algo que todos entendemos: se trata de una propiedad que atañe tanto a la sintaxis del lenguaje (el programa es una cadena que puede derivarse en la gramática) como a la semántica (el programa cumple con las especificaciones). No obstante, podemos objetar el mismo reparo terminológico que expresábamos en el tema "Lógica" (capítulo 4, apartado 2.4) con respecto a "fórmula bien formada": ¿acaso un "programa incorrecto" puede llamarse "programa"?

se mantenga en estas transformaciones. Una definición formal de la semántica es imprescindible para ello.

Finalmente, la definición semántica formal es útil en el proceso de diseño de nuevos lenguajes, para contrastar distintas posibilidades en la definición de las construcciones sintácticas básicas del lenguaje.

Como ya decíamos en el primer capítulo de este tema (apartado 3), se han propuesto varios enfoques para la definición formal de la semántica. Estos enfoques no son competitivos, sino complementarios: según el objetivo que se pretende con la formalización, resulta más adecuado un enfoque u otro. Resumimos a continuación las ideas básicas de los más conocidos.

4.2 Semántica operacional

La semántica operacional, también llamada interpretativa, se basa en una definición formal del intérprete del lenguaje, entendiendo "intérprete" en el sentido explicado al comienzo de este capítulo: una máquina que reconoce y ejecuta los programas escritos en ese lenguaje. Entonces, la semántica de cada construcción sintáctica se expresa mediante la descripción del funcionamiento de esa máquina virtual al interpretar la construcción.

Por ejemplo, para un lenguaje libre de contexto podemos definir un autómata de pila. El estado estará constituido por el estado de la parte de control y el estado de la pila, incluido el valor del puntero. Parte del estado está formada por el conjunto de los identificadores declarados en el programa y el valor que tengan. Esta parte se puede considerar como una ampliación del concepto de "evaluación" tal como se entiende en lógica: del mismo modo que la evaluación de variables proposicionales es una función que asigna a cada variable un valor del conjunto V , con $V = \{0,1\}$ en el caso de evaluación binaria (tema "Lógica", capítulo 2, apartado 3), y cuyo dominio se puede ampliar a las sentencias, podemos ahora definir una función, q , que asigne a cada identificador declarado en el programa un valor del conjunto de valores, y cuyo dominio se puede ampliar a las expresiones. El conjunto de estados sería el conjunto de todas esas funciones:

$$Q = \{q: \{\text{ident}\} \rightarrow V\}$$

donde $\{\text{ident}\}$ es el conjunto de identificadores y V es el conjunto de valores (enteros, reales, booleanos, etc.). Sobre esta función sería preciso hacer la restricción de que cada identificador sólo puede aplicarse en el subconjunto de valores determinado por su tipo.

Otra parte del estado sería la referente al estado de la pila: identificadores cuyo valor está almacenado, y posición del puntero de pila. El funcionamiento del autómata (que define la semántica del lenguaje) quedaría determinado especificando completamente la función de transición, $f(e,q)$, que depende del

estado actual, q , y de la entrada, e (sentencia o dato).

Veamos algunos ejemplos con relación al lenguaje definido en el apartado 3.4. Para no extendernos en demasiados detalles, nos referiremos sólo a la primera parte del estado (por lo que las definiciones serán incompletas). Para la sentencia de asignación podríamos escribir:

$$f(\text{ident} := \text{expr}, q_1) = q_2$$

donde, si $q_1(\text{ident}) = v_i$ y $q_1(\text{expr}) = v_e$, entonces

$$q_2(\text{ident}) = v_e \\ (\forall \text{ident}' \neq (\text{ident})(q_2(\text{ident}') = q_1(\text{ident}'))$$

Esta definición de la sentencia de asignación ya dice algo que no se suele decir en la definición informal: que la sentencia no tiene "efectos laterales", o sea, que no se modifica más que el valor de "ident". No obstante, está incompleta, y no sólo por el asunto de la pila, cuyo estado, eventualmente, tendría que modificarse, sino porque no indica lo que pasa cuando "ident" y "expr" son de diferente tipo. En nuestro lenguaje esto no importa, porque sólo hay un tipo ("integer"), pero en general, no es así, y entonces hay que completar la definición del lenguaje con otra función, "tipo", que se aplica sobre los identificadores y las expresiones y que toma valores en la interpretación de sentencias de declaración y de aquellas en las que hay evaluación de expresiones. También añadiríamos un estado especial de error, y completaríamos la definición de la sentencia de asignación formalizando la idea de que se va al estado de error si los tipos de "ident" y de "expr" son incompatibles. Si convenimos que la sentencia sólo es válida en el caso de que la evaluación de expr pertenezca al mismo tipo que ident, tendríamos:

$$f(\text{ident} := \text{expr}, q_1) = q_2 \text{ si } t(\text{ident}) \text{ definida y } t(\text{ident}) = t(\text{expr}) \\ = \text{error si } t(\text{ident}) \text{ indefinida o } t(\text{ident}) \neq t(\text{expr})$$

donde t es la función "tipo" a la que nos referíamos más arriba. Inicialmente, está indefinida para todo identificador. La función de transición le da valores para los identificadores al interpretar las sentencias de declaración y para las expresiones al interpretar sentencias donde intervengan tales expresiones.

Para la construcción que consiste en concatenar sentencias basta indicar que con la primera se va a un estado intermedio:

$$f(\text{sent1}; \text{sent2}, q) = f(\text{sent2}, f(\text{sent1}, q))$$

La semántica de la sentencia de iteración quedaría definida por la función de transición:

$$\begin{aligned}
 f(\text{while cond do sent, } q) &= f(\text{sent, } q) \text{ si } q(\text{cond}) = \text{verd} \\
 &= q \text{ si } (q(\text{cond}) = \text{falso}) \\
 &= \text{error en otro caso (por ejemplo, si } q(\text{cond}) \\
 &\quad \text{es un número)}
 \end{aligned}$$

Insistimos en que la definición completa exigiría una descripción también completa del autómata, incluyendo el estado de la pila. El principal inconveniente de la semántica operacional radica, precisamente, en su excesiva dependencia de una estructura de máquina (aunque ésta sea abstracta). La tendencia a buscar modelos más matemáticos y menos ligados a estructuras de máquinas conduce a la semántica denotacional.

4.3 Semántica denotacional

Se llaman *denotaciones* a entidades matemáticas que modelan los significados de las construcciones sintácticas del lenguaje. Así, escribiremos $S[\text{ident} = \text{expr}]$ para representar el *significado* de la sentencia de asignación. El principal problema para definir la semántica denotacional de un lenguaje determinado es la elección de las entidades matemáticas adecuadas. Veamos resumidamente cómo podríamos proceder en el caso de nuestro lenguaje del apartado 3.4.

Definimos primero los *dominios sintácticos*, que son los conjuntos de identificadores, I , de expresiones, X , y de sentencias u órdenes ("commands"), C . Normalmente, lo que en la definición sintáctica del apartado 3.4 llamábamos "condición" (que interviene en las sentencias "if" y "while") se suele considerar como perteneciente a la misma categoría sintáctica que "expresión", es decir, se consideran pertenecientes al mismo dominio las expresiones aritméticas y las booleanas. Definiremos a continuación los *dominios semánticos*, que son los conjuntos donde van a tomar valores las denotaciones.

El primer dominio semántico es V , el conjunto posible de valores. Así, $S[I]$ será una función que aplica cada identificador en su valor. En nuestro lenguaje sólo teníamos un tipo de datos, "integer", pero normalmente, V será la unión de los enteros, booleanos, caracteres, etc. Suponemos que V contiene también el elemento "indefinido".

Otro dominio semántico es el *estado*. Aunque no se haga referencia a ninguna máquina concreta, ni real ni virtual, es preciso, de todas formas, representar de algún modo el efecto de la ejecución. Podemos partir de una definición de estado similar a la de la semántica operacional (una función de I en V). Pero al objeto de definir también la semántica de las sentencias "read" y "write" vamos a incluir el estado de la entrada y de la salida:

$$E = S = V^*$$

Esto indica que E (y S) es el dominio de todas las posibles cadenas (incluyendo la vacía) de elementos de V . Entonces, definiremos el *dominio estado* como

$$Q = M \times E \times S$$

donde M es el *dominio memoria*, formado por todas las funciones de I en V :⁹

$$M = \{f: I \rightarrow V\}$$

o, escrito más abreviadamente,

$$M = I \rightarrow V$$

Es decir, el dominio estado, Q , es el conjunto de todas las triplas $q = (m, e, s)$, donde $m \in M$ es una función $I \rightarrow V$ y $e \in E$ y $s \in S$ son cadenas de V a la entrada y la salida.

Las denotaciones de los dominios sintácticos se definen mediante *funciones semánticas*, funciones que aplican dominios sintácticos en dominios semánticos. Así, las denotaciones de expresiones (consideramos que los identificadores son casos particulares de expresiones) son funciones de estados en valores (o sea, el significado de una expresión es que toma un valor que depende del estado):

$$S[x]: Q \rightarrow V \cup \{\text{error}\}$$

donde $x \in X$, y se ha considerado que la evaluación de una expresión puede dar lugar a un error (por ejemplo, si se intenta sumar un entero con un booleano). Las denotaciones de sentencias u órdenes son funciones de estados en estados (el significado de una sentencia es que cambia el estado, pudiendo también dar un error):

$$S[c]: Q \rightarrow Q \cup \{\text{error}\}$$

donde $c \in C$.

Utilizamos corchetes (además de paréntesis) para las funciones semánticas sólo para mejorar la legibilidad. Para representar, por ejemplo, la función S de la sentencia $A:=B$ sobre un estado $q = (m, e, s)$ escribiremos:

$$S[A:=B] q = \dots, \text{ o bien: } S[A:=B] (m, e, s) = \dots$$

⁹ El símbolo " \rightarrow " representa ahora la aplicación de un conjunto en otro (no el condicional de la lógica ni las reglas de producción).

Para un lenguaje determinado, se llaman *cláusulas semánticas* a las definiciones concretas de las anteriores funciones semánticas para todas y cada una de las construcciones sintácticas del lenguaje (dadas por la definición sintáctica). Veamos cómo podrían ser algunas de las cláusulas semánticas para nuestro lenguaje.

Empezando por las expresiones, la construcción más sencilla es $\langle \text{expr} \rangle ::= \langle \text{id} \rangle$. La correspondiente cláusula semántica es:

$$S[i](m, e, s) = \text{error si } m(i) = \text{indefinido} \\ = m(i) \text{ si no}$$

Es decir, el significado del identificador i es el valor que tenga definido en M , $m(i)$, y si está indefinido se produce un error.

Para $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$ definimos la cláusula:

$$S[e_1 + e_2] q = S[e_1] q + S[e_2] q \text{ si } \text{num}(S[e_1]) \text{ y } \text{num}(S[e_2]) \\ = \text{error si no}$$

donde num es un predicado que se aplica sobre el conjunto V y que es verdadero si el elemento es un número (entero o real) y falso si no. Tal como se definió la sintaxis de nuestro lenguaje esta condición no sería necesaria, al ser todos los valores enteros.

Veamos algunas cláusulas semánticas para sentencias. Para la de asignación tenemos:

$$S[i := x](m, e, s) = (m[v/i], e, s) \text{ si } S[x](m, e, s) = v \text{ y } \text{compat}(S[i](m, e, s), v) \\ = \text{error si no}$$

donde "compat" es un predicado binario sobre V que es verdadero si los tipos de sus argumentos son compatibles y falso si no, y $m[v/i]$ es la función

$$m[v/i](j) = v \text{ si } i = j \\ = m(j) \text{ si no}$$

que da a cada identificador el valor que tenía salvo a i , que le da v (resultado de la evaluación de $x : v = S[x](m, e, s)$); si ésta da error (o si los tipos son incompatibles), también lo da $y := x$.

Para la concatenación de sentencias,

$$S[c_1; c_2] q = \text{error si } S[c_1] q = \text{error} \\ = S[c_2](S[c_1] q) \text{ si no}$$

Para la sentencia "while",

$$\begin{aligned} S[\text{while } x \text{ do } c] q &= S[\text{while } x \text{ do } c] q' \text{ si } S[x] = v \text{ y } v = \text{verd y } S[c] = q' \\ &= q \text{ si } S[x] = v \text{ y } v = \text{falso} \\ &= \text{error en otro caso} \end{aligned}$$

En "otro caso" está comprendido el que $S[x]$ o $S[c]$ den error o que v no sea ni "verd" ni "falso". Obsérvese que la definición es recursiva.

Veamos, como último ejemplo, el caso de la sentencia de salida:

$$\begin{aligned} S[\text{write}(x)] m, e, s &= (m, e, s, v) \text{ si } S[x] = v \\ &= \text{error si no} \end{aligned}$$

donde " $s.v$ " es la cadena resultante de poner v detrás de s .

La semántica denotacional, de la que aquí sólo hemos presentado la idea básica, es una herramienta muy útil para el diseño de nuevos lenguajes. Sabiendo lo que se quiere del lenguaje, se diseñan denotaciones alternativas, se comparan, y cuando se está satisfecho con sus propiedades se les da nombre y se define la sintaxis. Para la verificación de programas escritos en un lenguaje ya definido, o para el desarrollo de métodos formales de diseño y prueba de los programas, resulta más útil la semántica axiomática.

4.4 Semántica axiomática

En la semántica operacional el estado es un concepto básico, ligado al concepto de máquina. En la semántica denotacional abstraemos el concepto de máquina, y consideramos el estado como un dominio semántico sobre el que se aplican las funciones semánticas, definidas sobre los dominios sintácticos. En la semántica axiomática sigue siendo básico el concepto de *estado*¹⁰, pero no se trata de establecer denotaciones de las construcciones sintácticas, sino de encontrar relaciones lógicas entre estados iniciales y finales que sirvan para expresar el significado de los programas, o, lo que es lo mismo, para decir formalmente lo que hacen. Como hablamos de propiedades de los estados y de relaciones entre estados, la base teórica para formalizar aquí la semántica es el cálculo de predicados.

Sabemos que un predicado (tema "Lógica", capítulo 4, apartado 1.3) es la formalización de una propiedad (predicado monádico) o de una relación (predicado poliádico) entre elementos de un "universo del discurso". Nuestro universo del discurso es el conjunto V de valores posibles que pueden tomar las variables, que, en nuestro caso, son los identificadores. Como el conjunto de

¹⁰ Si se nos presenta siempre como básico el concepto de estado es porque estamos considerando lenguajes imperativos. En un lenguaje lógico, por ejemplo, la semántica podría definirse como hicimos en el capítulo 4 del tema "Lógica"

todos los estados posibles es el conjunto de todas las funciones de identificadores en valores, un predicado aplicado a unos identificadores representa a un conjunto de estados. Por ejemplo, si i es un identificador y $M(i)$ es el predicado " i es mayor o igual que cero", $M(i)$ es falso para todos los estados en los que $\text{valor}(i) < 0$ y verdadero para todos los estados en los que $\text{valor}(i) \geq 0$. Por tanto, $M(i)$, o, más nemotécnicamente, " $i \geq 0$ ", representa el conjunto de estados tal que $\text{valor}(i) \geq 0$. Predicados de este tipo son *predicados sobre el estado*.

Lo que interesa es establecer *predicados sobre programas* y poder demostrar que son verdaderos. Si Q y R son predicados sobre estados y P es un programa, la notación

$$\{Q\} P \{R\}$$

significa: "si la ejecución de P comienza en un estado que satisface Q , entonces termina en un tiempo finito en un estado que satisface R ". Esto, realmente, es un predicado sobre P : es verdadero o falso según sea P . Lo que pretende la semántica axiomática es encontrar métodos para, a partir de Q y R (*especificaciones*), y dado un programa P , demostrar que el predicado $\{Q\} P \{R\}$ es verdadero, lo cual demuestra la corrección de P . O, también, dados Q y R , encontrar P tal que $\{Q\} P \{R\}$ sea verdadero (síntesis del programa).

Q es la *precondición*, o *aserción de entrada*. R es la *postcondición* o *aserción de salida*. Y como P está compuesto por una secuencia de sentencias, podemos considerar que cada sentencia va transformando un predicado en otro a partir de Q . Si P satisface al predicado $\{Q\} P \{R\}$, entonces el resultado de la última transformación será R . Tenemos así que definir, para cada sentencia y combinaciones de sentencias, un *transformador* de predicados sobre el estado.

Dados una sentencia S y un predicado R que representa el resultado de ejecutar S , se define otro predicado, $pmd(S, R)$, que representa al conjunto de *todos* los estados tales la ejecución de S a partir de uno cualquiera de ellos termina en un tiempo finito en un estado que satisface R . Entonces, escribir $\{Q\} S \{R\}$ es lo mismo que escribir $Q \rightarrow pmd(S, R)$, donde " \rightarrow " es ahora el condicional de la lógica (si...entonces). " $pmd(S, R)$ " es la *precondición más débil* de S con respecto a R .

Por ejemplo, si S es $i := i + 1$ y R es $i \leq 1$,

$$pmd(i := i + 1, i \leq 1) = i \leq 0$$

En efecto, para que tras la ejecución de $i := i + 1$ resulte un valor de $i \leq 1$ es necesario y suficiente que, antes de la ejecución, $i \leq 0$. La postcondición se cumple con cualquier precondición Q tal que $i \leq 0$ (como $Q = i \leq -1$, $Q = i \leq -2$, etc.), pero la más restrictiva de ellas (o *más débil*) es $i \leq 0$.

Obsérvese que pmd , además de ser un *transformador de predicados*, es también un predicado (puesto que su resultado es una aserción, verdadera o falsa, que representa al conjunto de estados en los que es verdadera) con dos argumentos: una sentencia, S , y un predicado, R . Estamos, pues, en lógica de predicados de segundo orden (tema "Lógica", capítulo 5, apartado 2.1).

La primera tarea, dada la definición sintáctica del lenguaje, es definir pmd para cada una de las posibles construcciones. Así, para la sentencia de asignación podemos definir:

$$pmd(i := e, R) = Rs$$

donde $s = \{e/i\}$ es la sustitución en R de e por i tal como fue definida en el tema "Lógica", capítulo 4, apartado 4.6. Esto quiere decir que la postcondición, R , será verdadera si y sólo si R con el valor de i sustituido por e es verdadera. O, lo que es lo mismo, que el conjunto de estados tras la ejecución de la sentencia es el inicial con el valor de i sustituido por el de e .

Para la sentencia condicional "ifthenelse",

$$pmd(\text{if } c \text{ then } S1 \text{ else } S2, R) = pmd(S1, R) \text{ si } c \text{ verdadera} \\ = pmd(S2, R) \text{ si } c \text{ falsa}$$

Para la concatenación de sentencias,

$$pmd(S1; S2, R) = pmd(S1, pmd(S2, R))$$

Para las sentencias iterativas (como "while") y de entrada y salida se hace precisa una elaboración más detallada en la que no vamos a entrar, y para la que, como siempre, remitimos a la bibliografía comentada en el apartado 7. Lo importante a recordar es que este enfoque de la semántica permite verificar si un programa P cumple o no unas especificaciones de entrada (Q) y salida (R), viendo si se satisface $\{Q\} P \{R\}$, y, yendo un poco más lejos, permite desarrollar reglas para sintetizar automáticamente P a partir de Q y R .

5. Procesadores de Lenguajes

5.1 Ensambladores

5.1.1 Traducción, carga y ejecución

Antes de entrar en el estudio de la estructura de un ensamblador conviene que nos detengamos un momento a ver su función en un contexto global, describiendo a grandes rasgos los procesos que tienen lugar desde la traducción hasta la ejecución de un programa.

Por razones ya explicadas tanto en este tema como en el de "Algoritmos", el desarrollo de un programa suele hacerse descomponiéndolo en módulos que se prueban independientemente. Por ello, la mayoría de los ensambladores no generan directamente un programa en lenguaje de máquina, sino lo que se llama *módulo objeto*. Este módulo sólo difiere del programa final en lenguaje de máquina en que las direcciones simbólicas no se han sustituido aún por las direcciones absolutas de memoria, sino que figuran en una tabla junto con sus desplazamientos con relación al comienzo del programa (como si éste fuera a cargarse a partir de la dirección 0). Además, puede haber direcciones simbólicas, conocidas como *referencias externas*, que corresponden a llamadas a otros módulos o subprogramas que se ensamblan aparte o que son rutinas de utilidad ya incluidas en el sistema operativo, como las de comunicaciones con los periféricos. Cuando se han ensamblado todos los módulos, un programa llamado *montador de enlaces*, o *enlazador* ("linker"), "resuelve" estas referencias externas y genera un *módulo de carga*. Finalmente, otro programa, el *cargador reubicador* ("relocating loader") asigna las direcciones absolutas e introduce en la memoria el programa en lenguaje de máquina listo para ser ejecutado. La figura 5.7 ilustra esta secuencia de procesos. Los rectángulos representan programas que se ejecutan, y los bloques ovalados datos o resultados de esos programas. Como se indica en la figura, los "módulos objeto" pueden haber sido generados por un ensamblador o por un compilador.

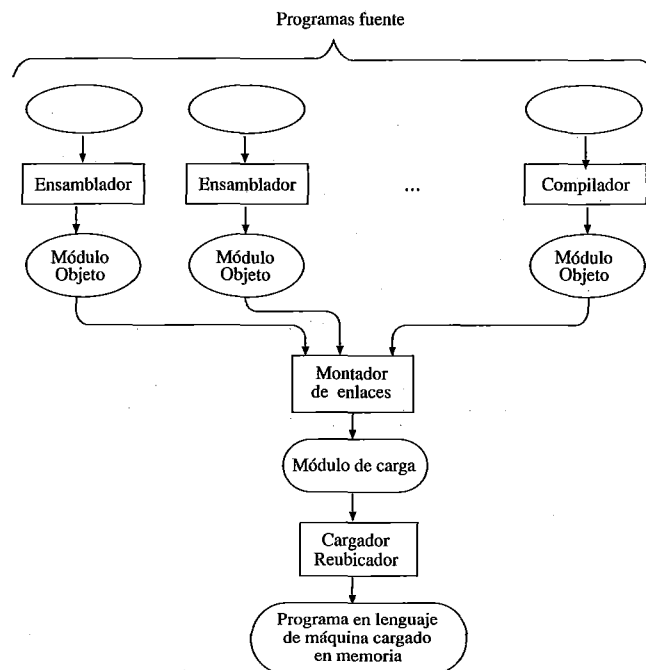


Figura 5.7.

Por limitaciones de capacidad, normalmente no es posible que coexistan en la memoria central todos los programas que intervienen: programas fuente, módulos objeto, ensamblador, compilador, montador y cargador. Por esta razón, durante el ensamblaje (o la compilación) de cada módulo sólo está presente, además de una parte del sistema operativo llamada *residente* (que incluye al cargador), el ensamblador (o el compilador). Este lee las instrucciones del programa fuente, una detrás de otra, de un periférico (normalmente, de un disco), y deposita el módulo objeto en algún medio de almacenamiento auxiliar (normalmente, un disco). Cuando todos los módulos objeto están disponibles, se carga el montador de enlaces, que genera el programa reubicable. La figura 5.8 insiste en esta secuencia de procesos. No debe interpretarse erróneamente el hecho de que en la figura aparezca cinco veces el símbolo del disco: los diferentes programas pueden estar en varios discos o en uno solo, en distintas zonas o ficheros.

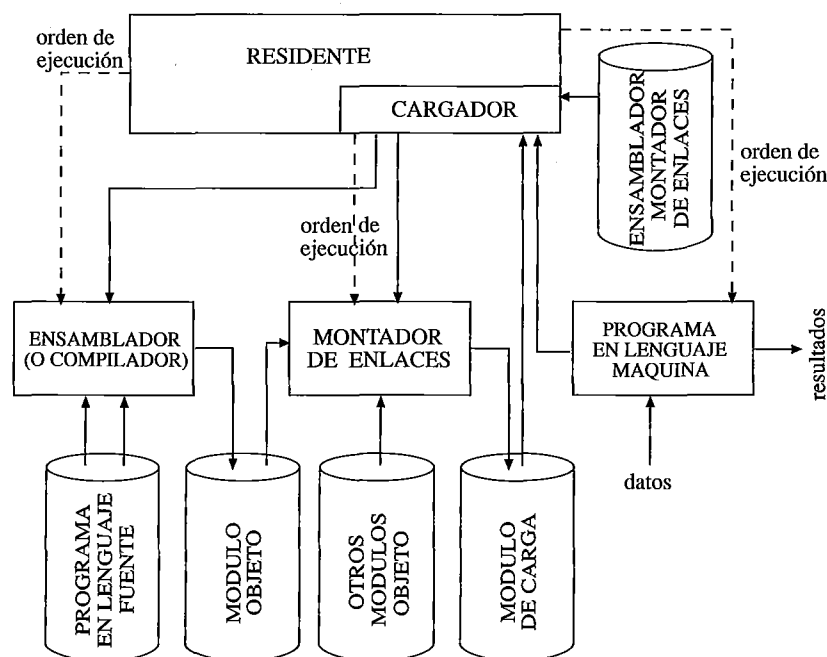


Figura 5.8.

5.1.2 Ensambladores de un paso y de dos

La mayoría de los ensambladores son de dos pasos. Esto quiere decir que leen dos veces el programa fuente. El motivo es que las instrucciones pueden

hacer referencia a etiquetas que aparecen más adelante. Por ejemplo, consideremos el siguiente segmento de un programa escrito en el lenguaje ensamblador que hemos descrito en los apartados 2.2.2 y 3.3:

	SAI	PRINCIP
UNO	CEN	1
N	MEM	1
A	MEM	100
PRINCIP	SAS	CALCULO
	...	
	...	

El ensamblador ha de generar un módulo objeto en el que la primera instrucción tiene que llevar como dirección el número que corresponda a la dirección de la etiqueta PRINCIP relativa al comienzo del programa. Es decir, a "SAI PRINCIP" le corresponde la dirección 0, a "UNO CEN 1" la 1, a "N MEM 1" la 2, y como después se reservan cien direcciones, a "PRINCIP SAS CALCULO" le corresponderá la dirección 103. Pero cuando el ensamblador lee la primera instrucción aún no puede saber qué dirección le va a corresponder a "PRINCIP". Por ello, en una primera lectura del programa fuente (*primer paso*) el ensamblador construye en la memoria una tabla de etiquetas, que en el caso anterior quedaría así:

<i>Etiqueta</i>	<i>Dirección</i>
UNO	1
N	2
A	3
PRINCIP	103
...	...
...	...

Cuando ha leído todo el programa y construido la tabla de etiquetas, entra en el *segundo paso*, leyendo de nuevo desde la primera línea. Al encontrarse con "SAI PRINCIP", consulta la tabla de etiquetas, donde ve que a "PRINCIP" le corresponde la dirección relativa 103, con lo que genera ya el código objeto correspondiente a esta instrucción. Desde luego, puede haber etiquetas que no aparezcan en la tabla, que corresponderán a referencias externas, de las cuales se ocupará el montador de enlaces.

Ahora bien, en ciertos casos no conviene que el ensamblador tenga que leer dos veces el programa fuente. Por ejemplo, en pequeños sistemas en los que no se dispone de unidades de disco: si el programa fuente no cabe en la memoria, sería preciso que el usuario introdujese manualmente dos veces ese pro-

grama fuente. Este caso es ciertamente raro en la actualidad (hay "estaciones de trabajo" sin disco, pero es porque utilizan un "servidor de ficheros" de la red a la que están conectadas). Otra motivación para eludir la doble lectura del programa fuente es tratar de reducir el tiempo total de proceso de ensamblaje. Para tales casos, se puede tener un ensamblador de un solo paso. Basta con ir formando una tabla adicional con los nombres no referenciados aún en la tabla de etiquetas y la dirección relativa de las instrucciones en las que aparecen; conforme se va llenando la tabla de etiquetas, se retrocede para poder obtener el código de las instrucciones pendientes de traducir.

5.1.3 Un ensamblador de dos pasos

Vamos a describir (sólo con organigramas, sin descender a la codificación) un posible ensamblador para el lenguaje ensamblador definido en los apartados 2.2.2 y 3.3. Como el proceso es bastante sencillo, no nos preocuparemos por hacer un uso sistemático de la definición formal del lenguaje dada en el apartado 3.3. Los organigramas no son detallados ni completos, ni siquiera estructurados¹¹, pero esperamos que permitirán al lector hacerse una idea concreta del proceso de ensamblaje.

La tarea del primer paso consiste, esencialmente, en formar la tabla de etiquetas, llevando, al mismo tiempo, algunas tareas de detección de errores sintácticos, como etiquetas erróneamente formadas o duplicadas. Para formar la tabla se utiliza una variable llamada *contador de ensamblaje*, que se inicializa en cero y se incrementa, normalmente en una unidad, al leer cada instrucción o seudoinstrucción. En el caso de la seudoinstrucción "MEM", se incrementa en el número de palabras a reservar. Así, al leer una línea, primero se detecta si hay algún carácter distinto de "bl" y de "*" en la primera columna; si es así, es que hay una etiqueta, cuyos caracteres se leen y se introducen en la tabla de etiquetas junto con el contador de ensamblaje. Se analiza luego el código de operación, y si éste es "MEM", se mira en cuánto hay que incrementar el contador de ensamblaje, tras lo cual se lee la siguiente línea, y así sucesivamente hasta encontrar la seudoinstrucción "FIN".

En la figura 5.9 puede verse el organigrama para este primer paso. `CONT_ENS` es una variable entera que corresponde al contador de ensamblaje. `LIN` es una variable estructurada como tira de caracteres. Si, por ejemplo, el número máximo de caracteres de cada línea es ochenta, la declaración de `LIN` en un programa en Pascal podría ser:

```
var LIN: array [1..80] of char
```

¹¹ Hemos decidido presentarlos así en aras de una mayor concisión. Comprendido el proceso, el lector puede tratar de presentarlo de manera estructurada, siguiendo los principios explicados en el tema "Algoritmos".

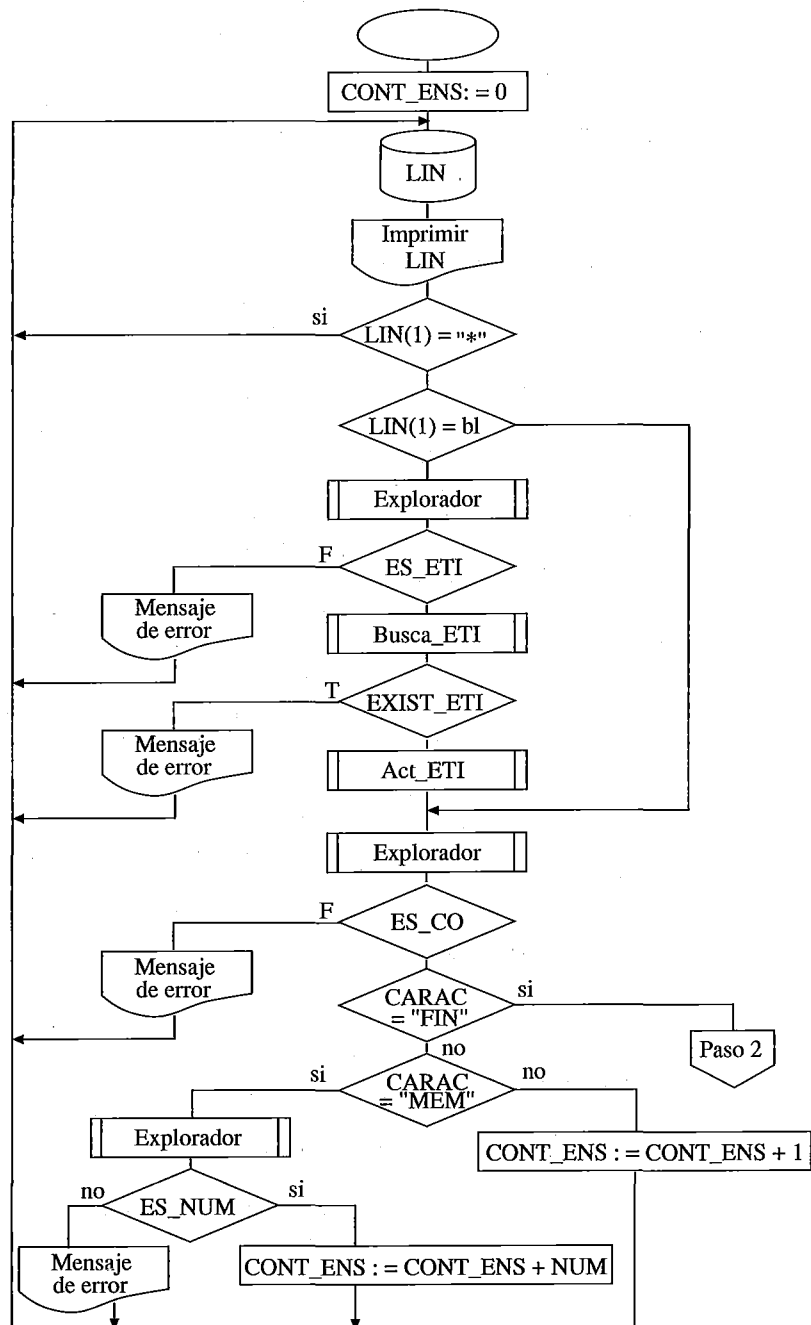


Figura 5.9.

Se supone que el programa fuente está almacenado en un fichero de un disco, de donde se van leyendo sucesivamente las líneas, que se introducen en la variable `LIN`.

En el organigrama aparecen llamadas a tres procedimientos, "Explorador", "Busca_ETI" y "Act_ETI", cuyas funciones son las siguientes:

"Explorador" recorre caracteres en la variable `LIN` saltando los espacios en blanco hasta encontrar el primero no blanco. Sigue entonces explorando hasta que de nuevo aparece un blanco. La secuencia de caracteres no blancos la pone en la variable `CARAC`. Si se trata de un número, lo convierte a binario, lo deja en la variable `NUM` y pone la variable booleana `ES_NUM` con el valor "true". Si es una etiqueta correctamente formada pone a "true" la variable booleana `ES_ETI`. Y si es un código de operación pone a "true" la variable booleana `ES_CO` y devuelve el código binario (cuatro bits) en la variable `CO`. alguna de estas informaciones, concretamente `NUM` y `CO`, no son necesarias en el primer paso, pero sí en el segundo. Aunque no se ha indicado en el organigrama, cada vez que el programa lee una línea nueva y llama a "Explorador" tiene que inicializarlo para que empiece a explorar desde el primer carácter de la línea; en llamadas sucesivas dentro del análisis de la misma línea, el explorador seguirá desde el carácter en que se detuvo (el primer blanco siguiente a una cadena de caracteres). En el apartado 5.2.2 explicaremos, en un contexto más general, una técnica para diseñar programas exploradores.

"Busca_ETI" es un subprograma que consulta la tabla de etiquetas: compara `CARAC` con las etiquetas ya introducidas. Si encuentra coincidencia, pone a "true" la variable booleana `EXIST_ETI` y devuelve en la variable entera `DIR` la dirección relativa correspondiente (esto último, para el segundo paso).

"Act_ETI" construye y actualiza la tabla de etiquetas: cada vez que se le llama introduce `CARAC` y `CONT_ENS` en la tabla.

El segundo paso puede verse en la figura 5.10.

Los códigos objeto de las distintas instrucciones y pseudoinstrucciones se van guardando sucesivamente en `MEM(CONT_ENS)`. En el organigrama aparece un mensaje de error en el caso de que una dirección simbólica no aparezca en la tabla de etiquetas. Esto es porque suponemos que nuestro ensamblador no permite el uso de referencias externas. Para permitirlo bastaría con que, en lugar de dar un error, se fuese construyendo otra tabla con la etiqueta no encontrada y la dirección relativa de la instrucción en la que aparece, y esta tabla se guardaría al final en el disco y serviría de entrada para el montador de enlaces.

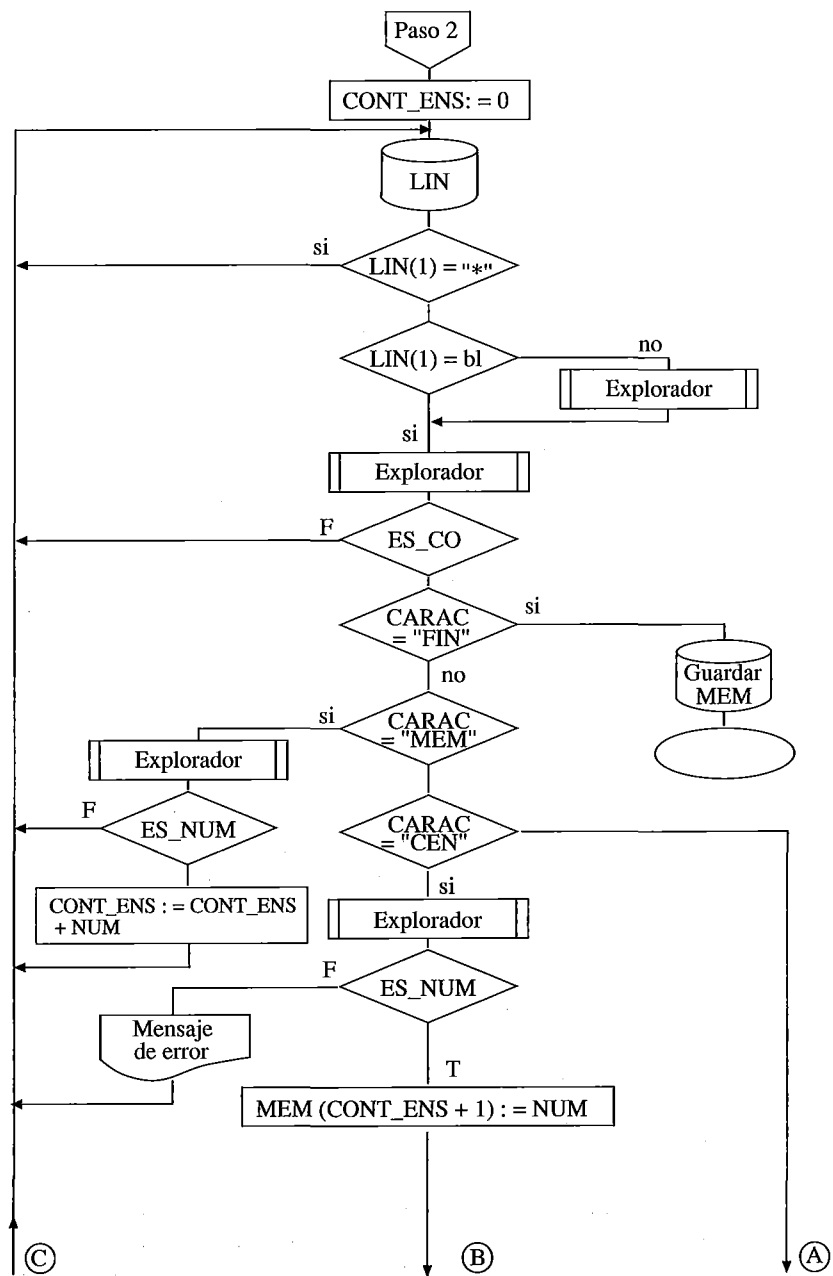


Figura 5.10.

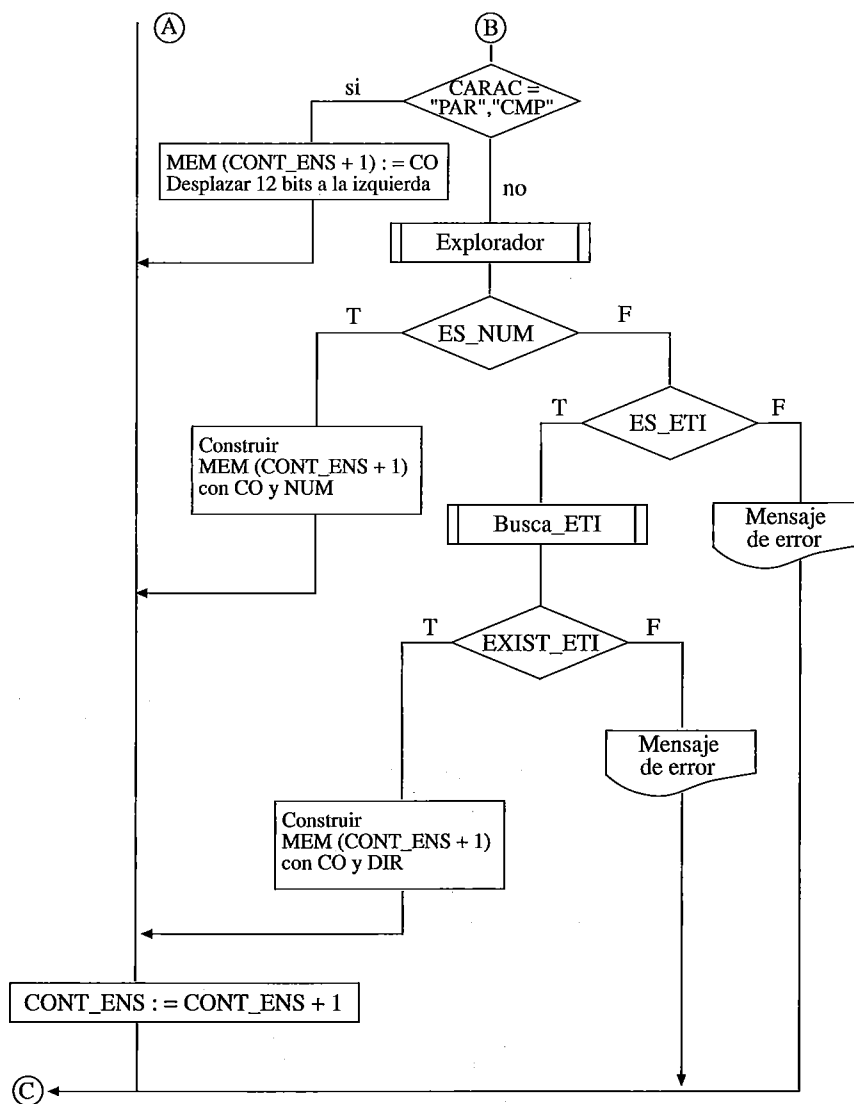


Figura 5.10. (Cont.)

Cuando se identifica un código de operación, se sintetiza la instrucción en lenguaje de máquina superponiendo el código binario en los cuatro primeros bits y a continuación la dirección, salvo si la instrucción no hace referencia a memoria (PAR, CMP, etc.), en cuyo caso basta con poner el código de operación.

El ensamblador para un ordenador real, con diversos registros y modos de direccionamiento, instrucciones de longitud variable, etc., puede ser algo más

complicado en sus detalles, pero seguirá teniendo, en esencia, la misma estructura que hemos explicado para este modelo sencillo.

5.2 Compiladores

5.2.1 Fases de la compilación

En la ejecución de un programa compilador podemos distinguir dos etapas: la de análisis, en la que se reconoce el programa fuente y se le traduce a un código o lenguaje intermedio, y la de síntesis, en la que se genera el módulo objeto. En cada etapa tienen lugar varias *fases*, como indica la figura 5.11.

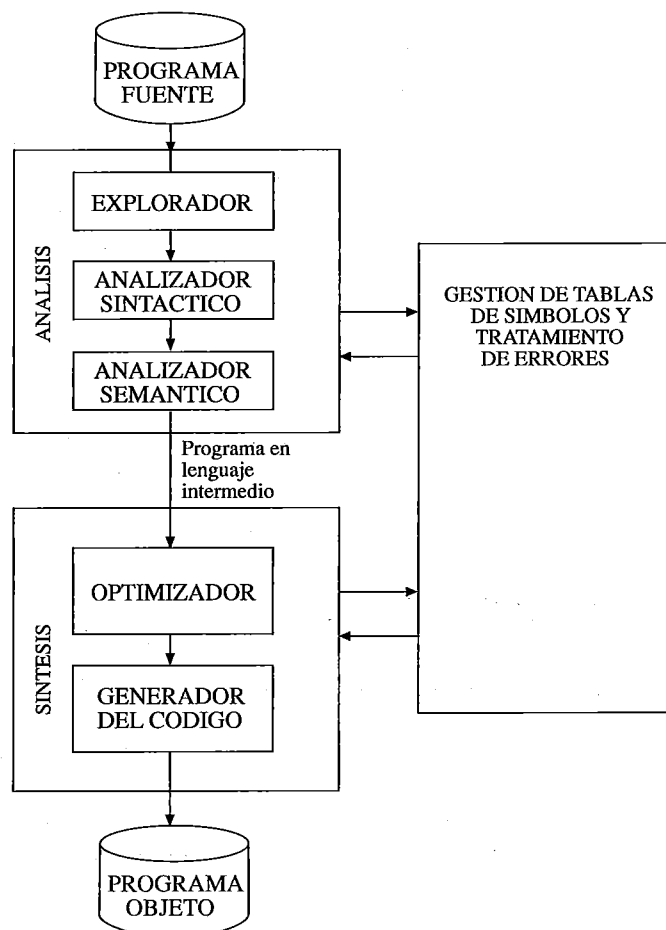


Figura 5.11.

El número de *pasos* del compilador (lecturas sucesivas del programa fuente o de alguna versión intermedia) es muy variable y depende de diversos factores condicionantes del diseño. Si la capacidad de memoria central es reducida, interesa realizar el proceso en muchos pasos, con resultados intermedios en memoria auxiliar. En el caso extremo, que mencionábamos al hablar de los ensambladores, de que no se disponga de memoria auxiliar, o cuando se quiera una gran velocidad de compilación, puede llegarse a realizar todo el proceso en un solo paso.

En este apartado describiremos a grandes rasgos las funciones de cada fase, y en los siguientes concretaremos viendo cómo se puede aplicar la definición formal de nuestro "minilenguaje" del apartado 3.4 para el diseño de las dos primeras.

La primera fase del análisis es el *análisis léxico*. El analizador léxico, al que también llamaremos *explorador* ("scanner")¹², tiene como función principal leer los caracteres del programa fuente e identificar las *categorías sintácticas* ("tokens") del lenguaje. Son categorías sintácticas las palabras clave (*begin*, *if*, ...), los identificadores, las constantes, etc. Recuérdese que en nuestro ejemplo de ensamblador del apartado anterior también teníamos un explorador que reconocía las etiquetas, los códigos de operación y los números. Además de esta labor de reconocimiento, el explorador va creando una *tabla de símbolos* en la que se registran los nombres y los atributos de los identificadores (tipo de variable, número y tipos de los argumentos en el caso de que sea un nombre de procedimiento, etc.).

La segunda fase es el *análisis sintáctico* ("parsing"). El *analizador sintáctico* ("parser") es, formalmente, un reconocedor del lenguaje. Como ya hemos dicho, los lenguajes de programación se suelen formalizar con gramáticas libres de contexto, y como sabemos del capítulo 4 (apartado 4), para reconocer lenguajes libres de contexto se precisan autómatas de pila. Y en efecto, durante la fase de análisis sintáctico se construyen estructuras de pila en la memoria central. Pero el analizador sintáctico no sólo reconoce si el programa es correcto o no, también reconstruye el árbol de derivación. Para su diseño resulta de gran utilidad disponer de la definición sintáctica formal.

Estas dos fases, como las siguientes, son fases lógicas que no necesariamente tienen que ejecutarse una tras otra. Así, el explorador es normalmente un subprograma llamado por el analizador sintáctico, como indica la figura 5.12.

Se puede pensar que, puesto que la definición formal del lenguaje permite formar programas (y, a la inversa, analizarlos) a partir de los caracteres elementales, ambas fases, análisis léxico y sintáctico, podrían fundirse en un solo algoritmo. Tal cosa es, en efecto, posible, pero hay varios motivos para hacer la separación. Aparte de facilitar un desarrollo modular, la tarea de análisis

¹² En otros textos se restringe el término "explorador", o "scanner", a un subprograma del analizador léxico que realiza las funciones más sencillas, como leer caracteres y eliminar espacios en blanco y cambios de línea.

léxico puede modelarse, como veremos con un ejemplo en el apartado 5.2.2, mediante un reconocedor finito, lo que conduce a programas muy eficientes para esa tarea.

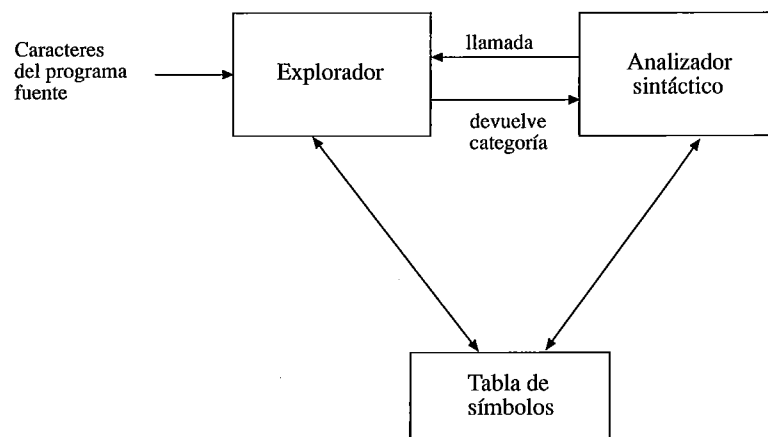
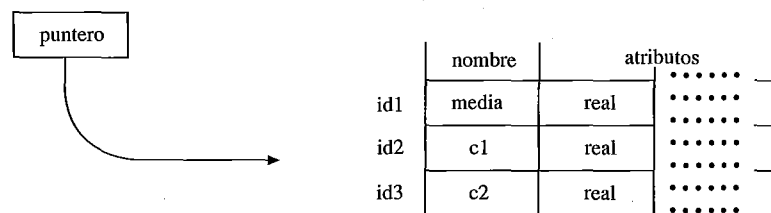


Figura 5.12.

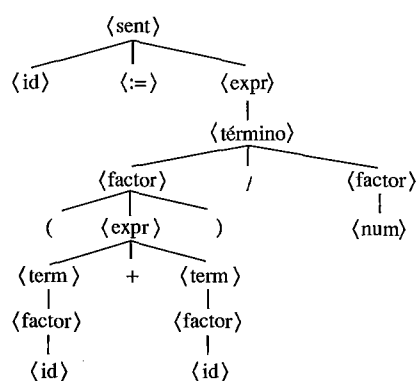
Concretemos estas funciones de los analizadores léxico (explorador) y sintáctico con un sencillo ejemplo: al analizar la sentencia

```
media := (c1 + c2)/2
```

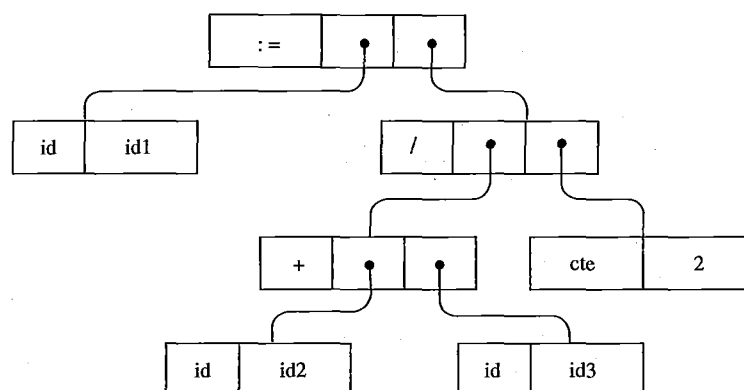
el explorador, en llamadas sucesivas del analizador sintáctico, reconoce "media", "c1" y "c2" como identificadores, ":", "=", "+" y "/" como símbolos terminales, con significado especial, del lenguaje, y "2" como una constante entera. Al consultar la tabla de símbolos, si los identificadores se han declarado previamente, los encuentra y devuelve sus punteros al analizador sintáctico (figura 5.13(a)). Este construye el árbol de derivación de la figura 5.13(b). La representación interna de este árbol puede hacerse mediante una estructura de datos como la de la figura 5.13(c), donde los rectángulos representan registros ("records") cuyo primer campo es el nombre de la categoría sintáctica, y en los otros campos puede haber punteros. Así, en cada nodo de identificador el puntero contiene la dirección de la tabla de símbolos en la que el explorador ha guardado el nombre y los atributos. En el nodo "cte" el siguiente campo contiene el valor numérico de la constante.



a) tabla de símbolos



b) árbol de derivación



c) estructura de datos

Figura 5.13.

La fase de *análisis semántico*¹³ tiene dos funciones. Una es la de interpretación, en el sentido de determinar el significado de las construcciones reconocidas por el analizador sintáctico, y de detectar los errores llamados semánticos (por ejemplo: asignación de un valor real a una variable declarada como booleana o uso de un identificador no declarado). La segunda función consiste en representar ese significado en un *lenguaje intermedio* entre el lenguaje fuente y el de máquina. Esta fase suele llevarse a cabo también en paralelo con el análisis sintáctico: conforme va generando el árbol de derivación, el analizador sintáctico va llamando a las correspondientes *rutinas semánticas*.

Para el ejemplo anterior, la traducción de la sentencia al lenguaje intermedio podría ser:

sum	id2	id3	m1
asign	2		m2
div	m1	m2	m3
trans	m3		id1

Este lenguaje sería el de una máquina virtual con tres direcciones de memoria. Las instrucciones de máquina (o "cuádruplas") tienen cuatro campos: un código de operación y las direcciones de dos operandos y del resultado. En las instrucciones de transferencia del contenido de una dirección a otra o de asignación de un valor constante a una dirección sólo se utilizan, como vemos, tres de los campos.

En la fase de *optimización* se procura modificar el código intermedio para obtener un programa que se ejecute más eficazmente (más rápido y usando menos recursos; por ejemplo, en el segmento de código anterior las dos instrucciones últimas pueden fundirse en una al tiempo que se ahorra una posición de memoria: `div m1 m2 id1`). Esta fase puede tener más o menos importancia dependiendo del uso que se vaya a hacer del compilador. Si el programa final compilado va a ejecutarse con mucha frecuencia y rara vez se va a modificar, entonces interesa que esté lo más optimizado posible, a costa de que el tiempo de compilación pueda ser muy grande. Si, por el contrario, se está desarrollando y probando un programa, interesa más bien que el tiempo de compilación no sea excesivo aunque el código final no esté muy optimizado.

En la fase final de *generación de código* se obtiene el programa traducido ya al lenguaje objeto. Algunos compiladores terminan con el programa en ensamblador, y otros incluyen las funciones de ensamblaje en la generación de código, de manera que producen un módulo objeto.

Así, el código en ensamblador, optimizado, para el ejemplo que venimos tratando podría ser:

¹³ Se trata aquí de lo que se llama "semántica estática", no de la semántica considerada en el apartado 4. En su mayor parte, este análisis semántico es necesario porque, como ya hemos dicho, en los lenguajes suele haber características dependientes del contexto que no se modelan en la sintaxis.


```

M2      CEN      2
      ...

      CAR      ID2
      SUM      ID3
      DIV      M2
      ALM      ID1

```

donde ID1, ID2, ID3 son las direcciones relativas de la tabla de símbolos. Si no existiera la instrucción de dividir, se haría un salto a un subprograma de división.

5.2.2 Un analizador léxico

Veamos cómo podría diseñarse un explorador para el lenguaje definido en el apartado 3.4. Consideraremos categorías sintácticas, que el explorador tiene que reconocer e informar de su aparición al analizador sintáctico, a:

- a) los identificadores ("`<identif>`");
- b) los números enteros sin signo ("`<num>`");
- c) los operadores relacionales ("`<oper>`");
- d) los operadores aritméticos "+", "-", "*" y "/", y los símbolos terminales ":", ",", ".", "(", ")" y "=";
- e) las palabras clave "begin", "end", "const", "var", "integer", "if", "then", "else", "while", "do", "read" y "write".

Las palabras clave, así como el símbolo ":"=" y los operadores ">=", "<=" y "<>" se consideran símbolos terminales en la gramática definida en el apartado 3.4, pero para el explorador son cadenas, puesto que para él los símbolos terminales son los caracteres individuales.

Si hemos elegido estos símbolos auxiliares y terminales como categorías sintácticas es porque para cada uno de ellos podemos definir una gramática regular:

- a) Para `<identif>`, en lugar de la producción (5) del apartado 3.4 podemos escribir:

```

<id> ::= a | b | ... | z | A | B | ... | Z
<id> ::= a<restoid>|b<restoid>|.....|Z<restoid>
<restoid> ::= a | b | ... | Z | 0 | 1 | ... | 9
<restoid> ::= a<restoid>|b<restoid>|...|Z<restoid>|0<restoid>|...|9<restoid>

```

Para esta gramática tenemos el reconocedor finito (capítulo 4, apartado 5.3) de la figura 5.14(a), y el lenguaje puede representarse también mediante la expresión regular:

$$(a+b+\dots+A+B+\dots)(a+b+\dots+A+B+\dots Z+0+1+\dots+9)^*$$

b) Análogamente, para <num> podemos escribir:

$$\langle \text{num} \rangle ::= 0 \mid 1 \mid \dots \mid 9$$

$$\langle \text{num} \rangle ::= 0 \langle \text{restonum} \rangle \mid 1 \langle \text{restonum} \rangle \mid \dots \mid 9 \langle \text{restonum} \rangle$$

El reconocedor es el de la figura 5.14(b), y la expresión regular:

$$(0+1+\dots+9)(0+1+\dots+9)^*$$

c) Para <oper> las reglas regulares serían:

$$\langle \text{oper} \rangle ::= = \mid > \mid < \mid > \langle \text{restop1} \rangle \mid < \langle \text{restop2} \rangle$$

$$\langle \text{restop1} \rangle ::= =$$

$$\langle \text{restop2} \rangle ::= = \mid >$$

Y el reconocedor finito es el de la figura 5.14(c). (La expresión regular es, simplemente, la suma de las cadenas =, <, >, <=, >= y <> : el lenguaje es finito).

d) De igual modo, para los otros símbolos tenemos el reconocedor de la figura 5.14(d).

e) Finalmente, para cada una de las palabras clave es trivial encontrar las reglas regulares y el reconocedor (cada una de ellas es un lenguaje con una sola cadena).

Por ejemplo, para "begin" podríamos escribir:

$$\langle \text{begin} \rangle ::= b \langle \text{egin} \rangle$$

$$\langle \text{egin} \rangle ::= e \langle \text{gin} \rangle$$

$$\langle \text{gin} \rangle ::= g \langle \text{in} \rangle$$

$$\langle \text{in} \rangle ::= i \langle \text{n} \rangle$$

$$\langle \text{n} \rangle ::= n$$

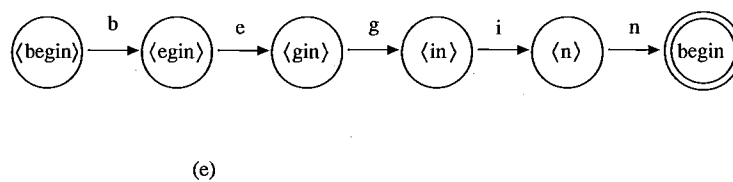
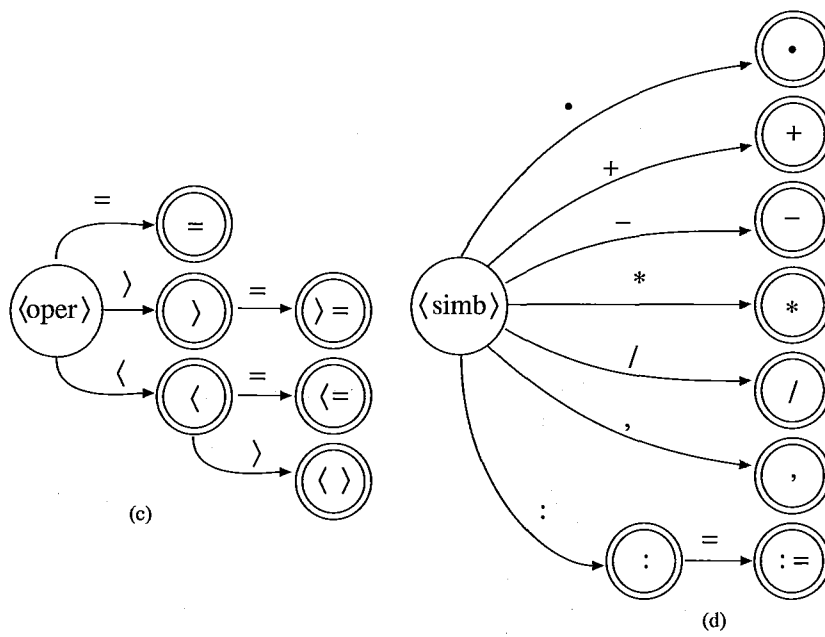
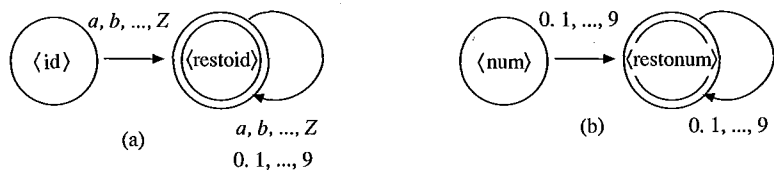


Figura 5.14.

Y el reconocedor sería el de la figura 5.14(e). No obstante, la consideración de todas las palabras clave implicaría un número muy elevado de estados, y, consecuentemente, muchas instrucciones para el programa explorador. Por ello, suele preferirse otra solución: las palabras clave se tratan, a efectos de reconocimiento, igual que los identificadores, pero inicialmente se introducen en la tabla de símbolos con un atributo que indica su categoría de palabra clave. Cuando el explorador reconoce un identificador tiene que explorar en cualquier caso la tabla de símbolos; si se trata de una variable o de una constante devuelve el puntero, y si está registrado como palabra clave, el código de la palabra.

Para diseñar el explorador con la ayuda de estos reconocedores tenemos que hacer algunas pequeñas modificaciones. Por una parte, como el explorador no puede saber a priori cuál es la categoría que va a reconocer en los caracteres que todavía no ha leído, fundiremos todos los diagramas en uno solo con un único estado inicial.

Por otra, el explorador irá transitando, a medida que lee caracteres, de un estado a otro, pero sólo puede terminar reconociendo una categoría cuando le llega el primer carácter que sigue a esa categoría (salvo si se trata de un solo símbolo, "*", "/", etc.). Por ejemplo, al explorar de izquierda a derecha la cadena

$$\text{media} := (c1+c2)/2$$

reconocerá "media" como identificador cuando vea que le sigue un espacio en blanco (o ":", si no se deja el espacio). Teniendo esto en cuenta, los estados finales no pueden ser los que aparecen en la figura 5.14, sino otros a los que se llega a partir de ellos cuando el último carácter leído permite clasificar la cadena de los anteriores en una u otra categoría. De este modo, nuestro explorador siempre irá "adelantado en un carácter".

El diagrama completo del reconocedor (finito y determinista) puede verse en la figura 5.15.

Hemos denominado a los estados con números, salvo los finales, representados por la categoría que devuelven al analizador sintáctico. Para simplificar, "/" representa "cualquier letra" y "d", "cualquier dígito". En las transiciones en las que no figura ningún carácter debe entenderse "cualquier otro carácter que no sea uno de los que corresponden a las otras transiciones que salen del mismo estado". Además, hemos incluido la función de ignorar espacios en blanco ("bl") y cambios de línea ("cl") (transición que entra en y sale del estado 1) y comentarios. Suponemos que los comentarios pueden aparecer en cualquier punto del programa fuente y se caracterizan porque comienzan con "(" (*) y terminan con "*)" ". Los estados 7 y 8 identifican el comienzo de un comentario, y cuando se reconoce el final (estado 9 con carácter ")") se vuelve al estado inicial.

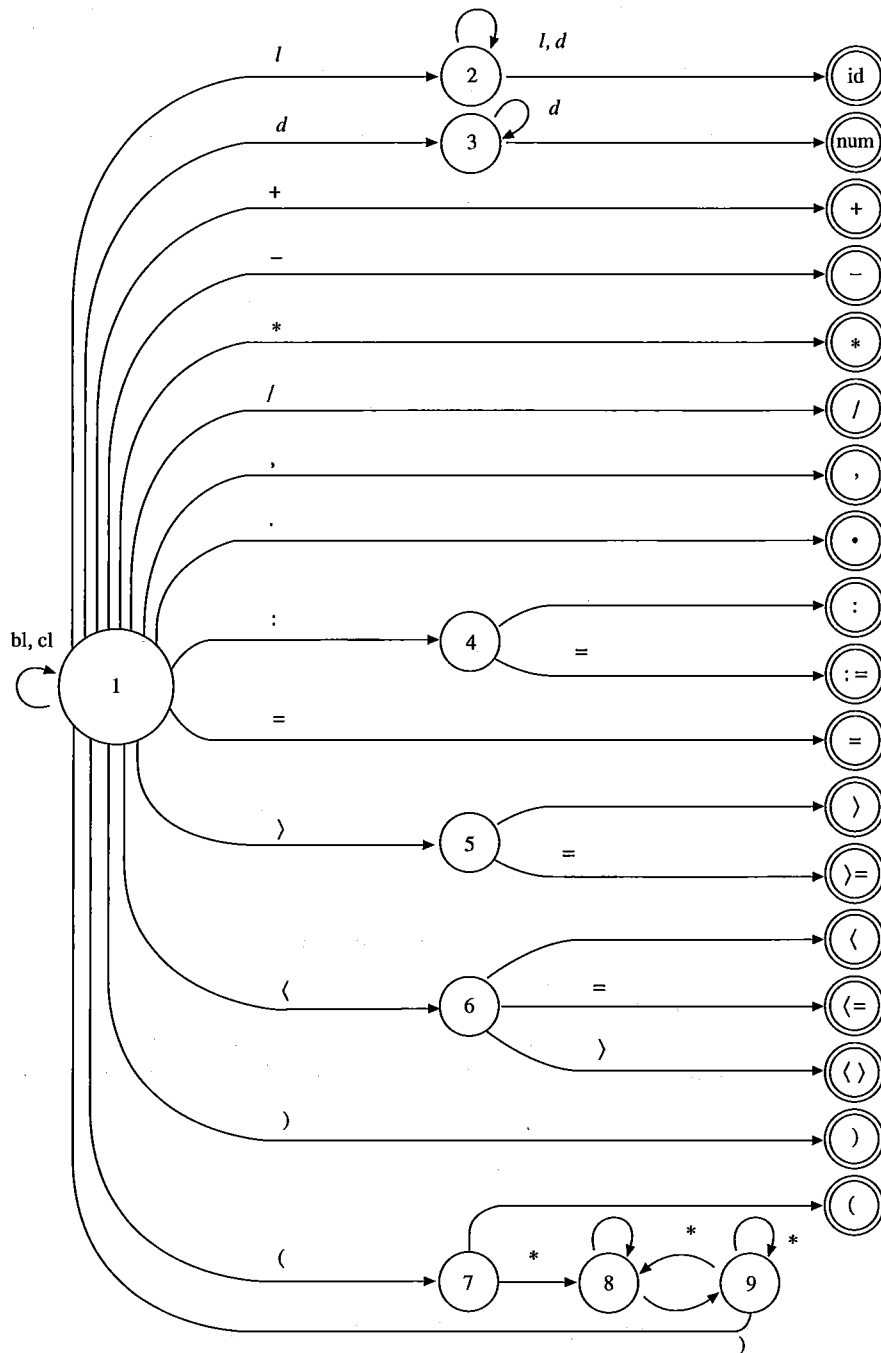


Figura 5.15.

A partir del diagrama, podemos escribir la tabla de transiciones del reconocedor (figura 5.16). Esta tabla puede estar en memoria y servir como guía al programa explorador. Para ello, podemos programar un procedimiento, al que llamaremos "transición" que reciba como parámetros de entrada el último carácter leído (car) y el número del estado anterior (est_ant), y que, tras consultar la tabla, devuelva como parámetro de salida el estado resultante (est_act).

	bl	cl	1	d	+	-	*	/	,	.	:	=	>	<	()
1	1	1	2	3	+	-	*	/	,	.	4	=	5	6	7)
2	id	id	2	2	id	id	id	id	id	id	id	id	id	id	id	id
3	cte	cte	cte	3	cte	cte	cte	cte	cte	cte	cte	cte	cte	cte	cte	cte
4	:	:	:	:	:	:	:	:	:	:	:	:=	::	:	:	:
5	>	>	>	>	>	>	>	>	>	>	>	>=	>	>	>	>
6	<	<	<	<	<	<	<	<	<	<	<	<=	<>	<	<	<
7	((((((8	(((((((((
8	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	8
9	8	8	8	8	8	8	9	8	8	8	8	8	8	8	8	1

Figura 5.16.

Otra posibilidad para programar el procedimiento "transición", sin necesidad de explorar ninguna tabla, es construirlo a base de sentencias condicionales (o de tipo "case") escritas siguiendo el diagrama de la figura 5.15:

```

if est_ant = 1 and car in letra
then est_act = 2
else ...

```

etc.

En cualquier caso, disponiendo del procedimiento "transición", el explorador puede programarse siguiendo el organigrama de la figura 5.17. El analiza-

dor sintáctico deberá leer un carácter antes de llamarlo la primera vez. Se supone que el procedimiento "lee_car" lee el siguiente carácter del programa fuente. "Act_Tab_Simb" consulta, si es necesario (o sea, en el caso de que el estado final sea "id" o "cte") la tabla de símbolos y devuelve los atributos al analizador sintáctico.

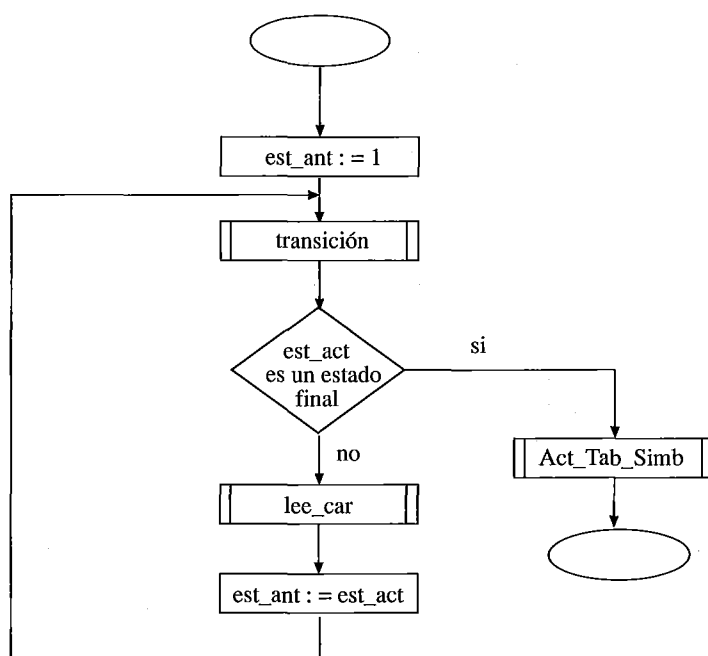


Figura 5.17.

5.2.3 Un analizador sintáctico

Existe una gran variedad de algoritmos de reconocimiento para gramáticas libres de contexto. En general, pueden clasificarse en *descendentes* y *ascendentes*. Los primeros construyen los árboles de derivación desde la raíz (símbolo inicial) hacia abajo (con nuestro convenio de representación, la raíz está arriba, al contrario de lo que ocurre con los árboles de la naturaleza). En los algoritmos ascendentes el análisis procede en sentido contrario: a partir de las categorías sintácticas, buscan el árbol ascendiendo hacia la raíz. Para una gramática concreta, es más fácil diseñar un algoritmo descendente (siempre que la gramática cumpla ciertas condiciones). Pero los algoritmos ascendentes

son más adaptables a distintos tipos de gramáticas. Por esta razón, los primeros son preferibles cuando se diseña "a mano" un compilador para un lenguaje, mientras que los segundos resultan más útiles en herramientas que generan algoritmos de análisis a partir de las especificaciones del lenguaje (apartado 5.5).

Veremos aquí, aplicado al minilenguaje definido en el apartado 3.4, el algoritmo más sencillo y más utilizado. Se trata de un algoritmo descendente recursivo llamado *predictivo*, que está basado en la definición sintáctica formal del lenguaje: para cada símbolo auxiliar se escribe un procedimiento de acuerdo con la producción que le corresponde. Como en el lado derecho de esa producción puede figurar el propio símbolo (u otro auxiliar en cuya producción figure el primero), los procedimientos pueden ser recursivos. Este algoritmo funciona "un símbolo por adelantado y sin retroceso" ("*one-symbol-look-ahead without backtracking*"). Esto quiere decir que, conforme se avanza en el análisis del programa fuente, la decisión sobre la acción a seguir se basa exclusivamente en el último símbolo (categoría sintáctica) entregado por el explorador (del mismo modo que éste entrega el resultado tras leer el carácter que le sigue). "Sin retroceso" significa que no vuelve atrás en el proceso de construir el árbol de derivación. Ahora bien, para que esto sea posible es preciso que la gramática cumpla las dos condiciones enunciadas en el apartado 3.2. Veámoslo con dos ejemplos.

Primer ejemplo: consideremos la gramática

$$\begin{aligned} S &::= A \mid B \\ A &::= aA \mid b \\ B &::= aB \mid c \end{aligned}$$

que no cumple la primera condición, porque a partir de A y B pueden derivarse cadenas que empiezan con el mismo símbolo terminal, " a ". Si tratamos de reconstruir el árbol de derivación de la cadena " ac ", primero leeríamos el símbolo " a ". La aplicación de la primera producción nos llevaría a la segunda, en la cual vemos que el símbolo coincide con el primero de " aA ". En este momento, el árbol de derivación construido es el de la figura 5.18(a). Avanzamos en la lectura, tratando de emparejar el resto de la cadena de entrada con " A ". Pero el resto es " c ", y no existe la producción " $A ::= c$ ". Por tanto, sería preciso retroceder para elegir " $S ::= B$ " y construir el árbol de la figura 5.18(b). Vemos que el problema está en que el símbolo leído, " a ", no determina unívocamente la producción a elegir. El lector puede comprobar que esta otra gramática:

$$\begin{aligned} S &::= A \mid aS \\ A &::= b \mid c \end{aligned}$$

es equivalente y no presenta ese problema.

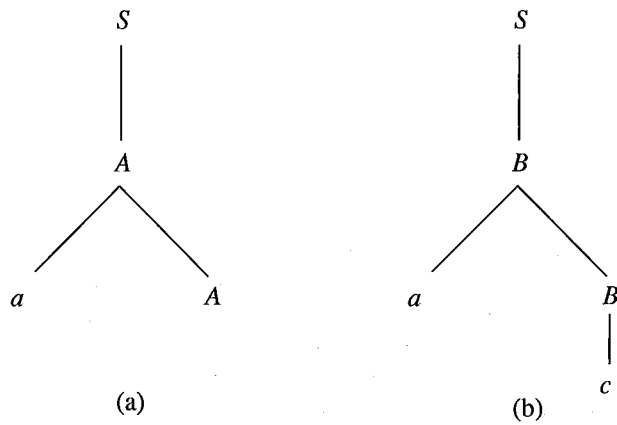


Figura 5.18.

Segundo ejemplo: la gramática

$$S ::= a \mid Sb$$

es recursiva por la izquierda, por lo que no cumple la segunda condición. Si tratamos de reconocer la cadena "abb", en cuanto se lee la primera "b" se produce un bucle infinito, como indica el árbol de la figura 5.19(a), en el que el procedimiento correspondiente a "S" se llama siempre a sí mismo sin seguir leyendo más caracteres de la cadena de entrada.

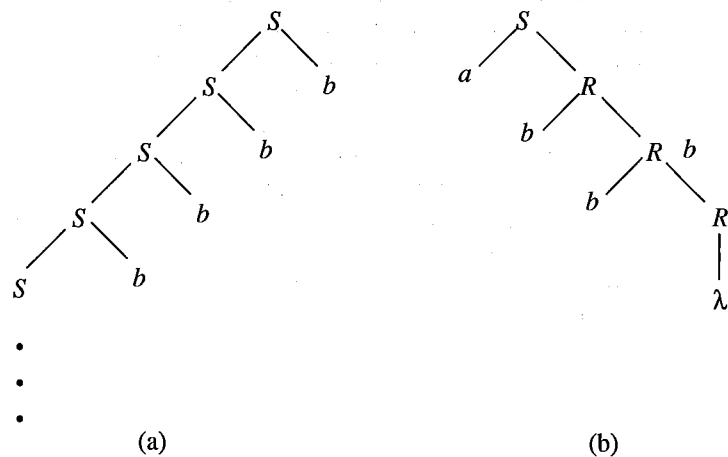


Figura 5.19.

El problema está en que habría que seguir leyendo hasta el final para decidir en qué momento aplicar la primera producción. Con la gramática equivalente

$$\begin{aligned} S &::= aR \\ R &::= bR \mid \lambda \end{aligned}$$

se puede generar, leyendo los símbolos de "abb" de izquierda a derecha y sin retroceder, el árbol de la figura 5.19(b).

Es fácil comprobar que la gramática de nuestro lenguaje cumple las dos condiciones. Veamos ahora un método mediante el cual puede obtenerse el algoritmo predictivo a partir de la definición formal (de las producciones, o, equivalentemente, del diagrama sintáctico) para cualquier gramática que cumpla esas condiciones.

El algoritmo en cuestión puede obtenerse a partir de los diagramas sintácticos siguiendo las normas resumidas en la figura 5.20. Para cada subdiagrama sintáctico del tipo que aparece a la izquierda se escribe un procedimiento de acuerdo con el organigrama de la columna central, o con el esquema de codificación en Pascal de la última columna. Explicamos las abreviaturas que se utilizan:

$P(\alpha)$ significa "procedimiento asociado al subdiagrama sintáctico de α ".

"categ" es la categoría sintáctica que devuelve el explorador cuando se le llama. Esto sucede, como indica la última parte de la figura, cuando en el diagrama sintáctico aparece un símbolo terminal (recuérdese que, para el analizador sintáctico, los símbolos terminales son las categorías sintácticas). Como el analizador va "un símbolo por adelantado", la última categoría entregada por el explorador tiene que coincidir con el símbolo del diagrama; entonces, se llama al explorador y se sigue adelante.

" $\text{prim}(\alpha)$ " representa el conjunto de todos los primeros símbolos terminales que pueden encontrarse en las cadenas derivadas a partir de α .

La primera de las condiciones que se han enunciado para la gramática se puede expresar también diciendo que si $A ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ es una producción, entonces $\text{prim}(\alpha_1) \cap \text{prim}(\alpha_2) \cap \dots \cap \text{prim}(\alpha_n) = \emptyset$.

Normalmente, el diagrama sintáctico muestra $\text{prim}(\alpha)$ muy claramente, porque α suele empezar por un símbolo terminal, y $\text{prim}(\alpha)$ es justamente ese símbolo.

Por ejemplo, en la ramificación que hay para "sentencia" en los diagramas de la figura 5.6,

$$\text{prim}(\alpha_1) = \text{begin}, \text{prim}(\alpha_2) = \text{identif}, \text{prim}(\alpha_3) = \text{if}, \text{etc.}$$

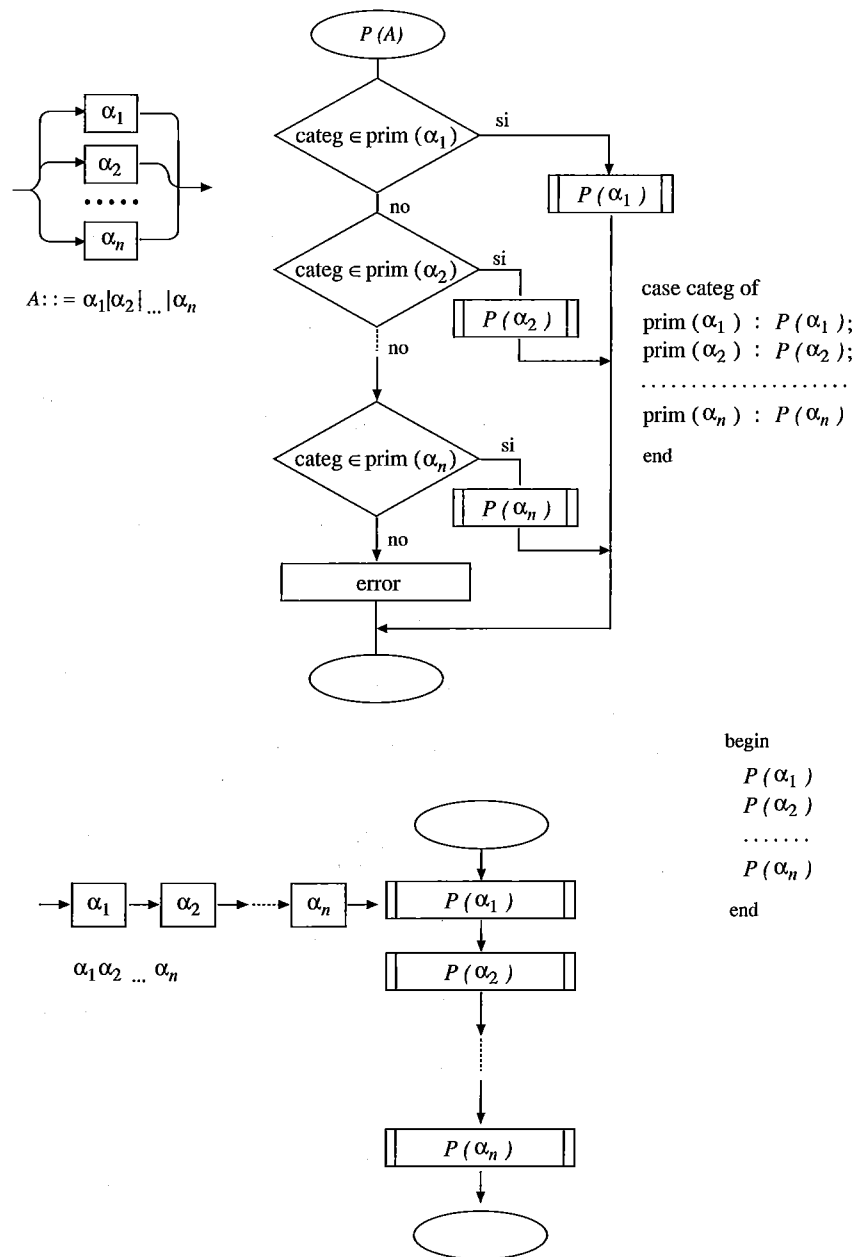


Figura 5.20.

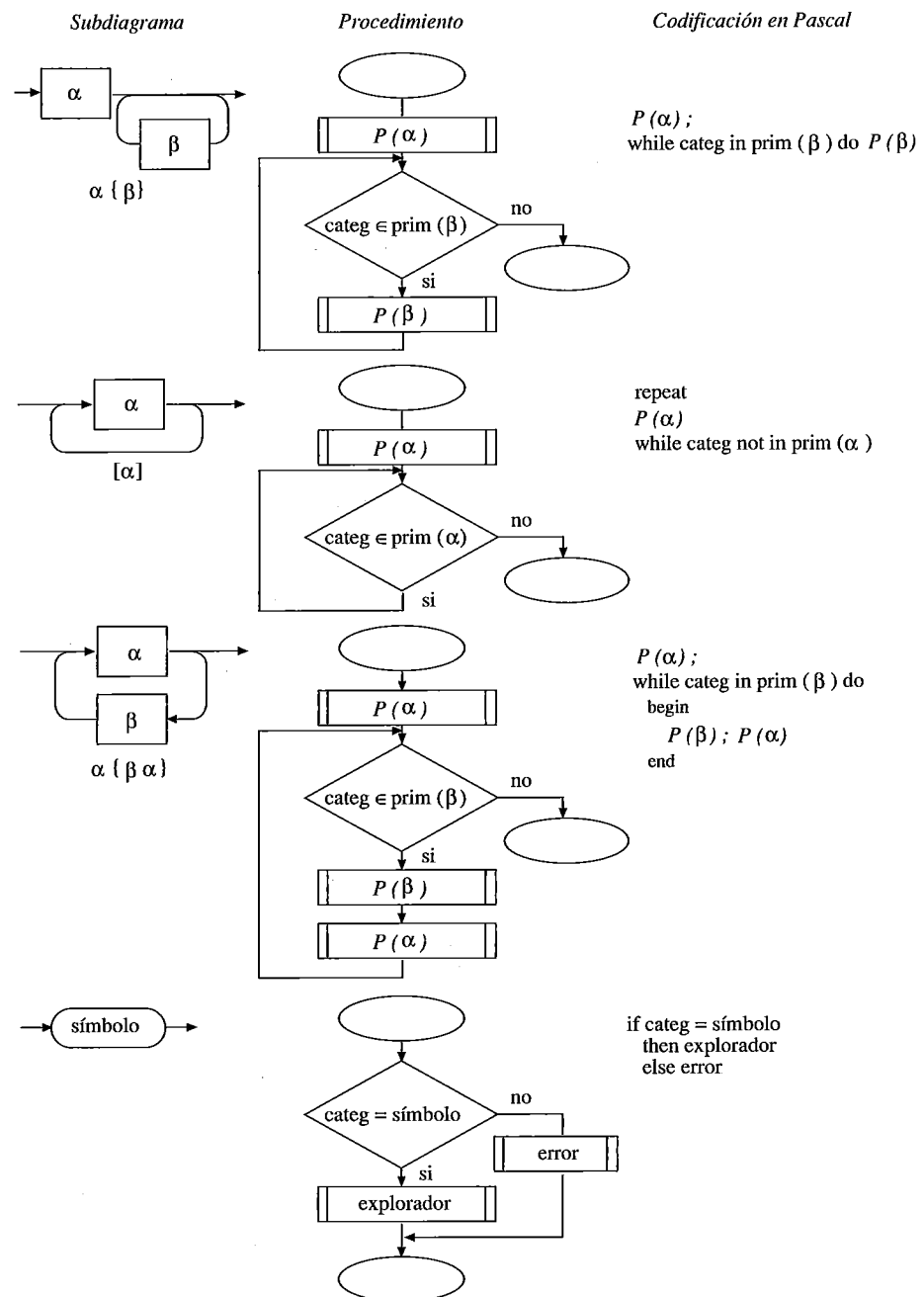


Figura 5.20. (Cont.)

La aplicación de estas normas al caso particular de nuestro lenguaje es inmediata. En la figura 5.21 puede verse el organigrama correspondiente al primer subdiagrama de la figura 5.6. Este podría ser el programa principal, a partir del cual se llama a "bloque", y de éste a los demás procedimientos. Antes de llamar la primera vez al explorador se inicializa la variable "car" con un espacio en blanco. Esto es necesario porque, como se recordará, el explorador funciona con "un carácter por adelantado".

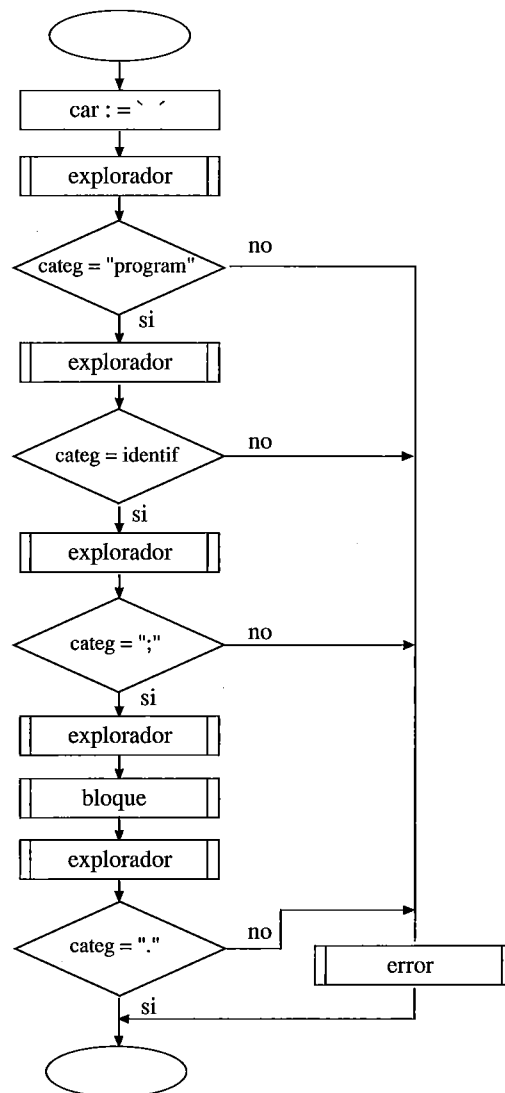


Figura 5.21.

El resto de los organigramas no presenta mayores problemas. A título de ejemplo, indicamos en las figuras 5.22, 5.23 y 5.24 los correspondientes a los procedimientos "expresión", "término" y "factor".

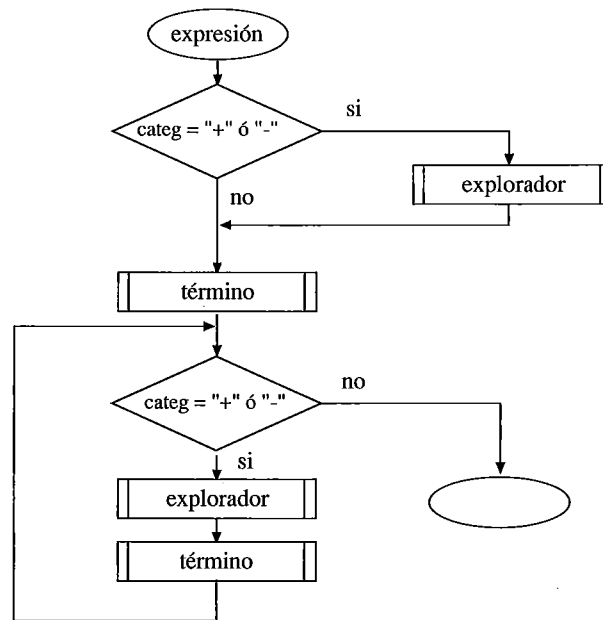


Figura 5.22.

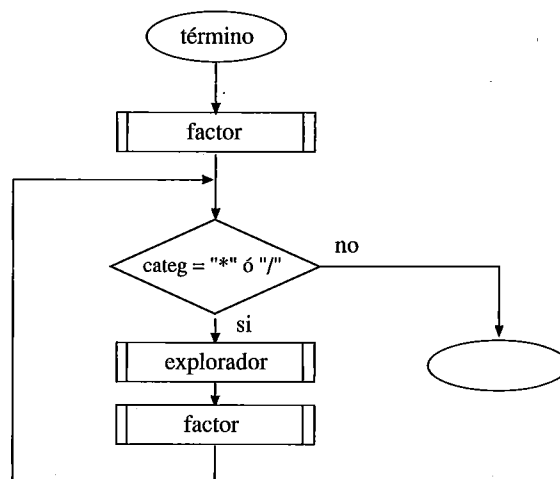


Figura 5.23.

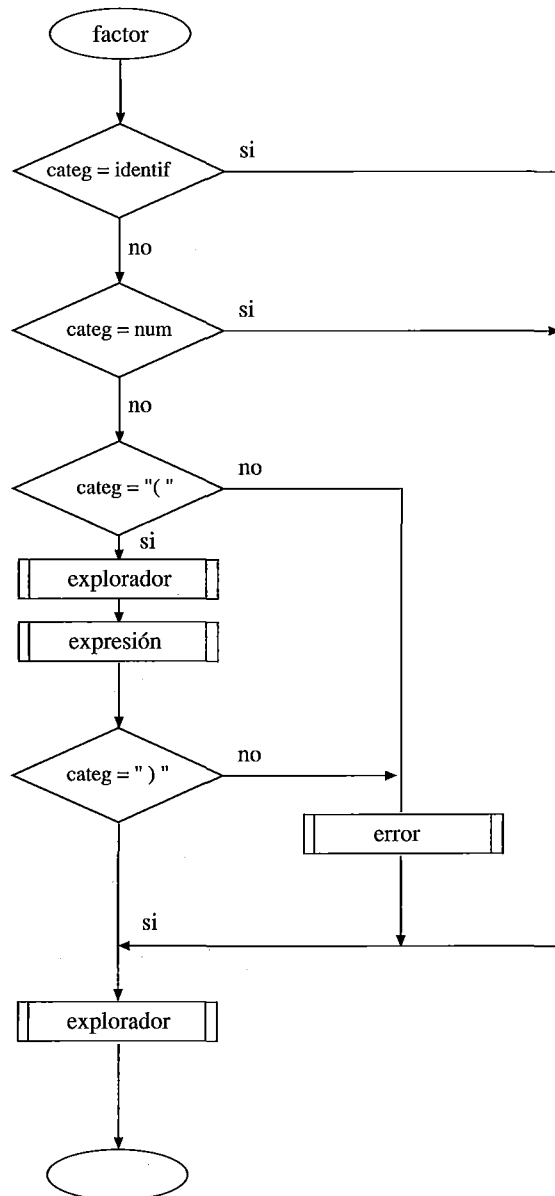


Figura 5.24.

En esta descripción nos interesaba sobre todo ver cómo la definición sintáctica formal del lenguaje es una herramienta útil para el diseño sistemático del procesador. Hemos prescindido de detalles, como el tratamiento de errores

y el manejo de la tabla de símbolos, que, aun siendo importantes, serían ya propios de un texto sobre compiladores.

Quizás el lector se pregunte por la construcción del árbol de derivación, que, como decíamos en el apartado 5.2.1, es una tarea del analizador sintáctico. Pues bien, en este algoritmo predictivo el árbol está implícito en la secuencia de procedimientos que se van llamando. Los punteros que aparecen en la figura 5.13 los va devolviendo el explorador conforme va reconociendo identificadores, y los utilizan las rutinas semánticas para ir generando el código intermedio. Estas rutinas semánticas son procedimientos asociados a cada uno de los procedimientos que realizan el análisis sintáctico. Por ejemplo, cuando se analiza la sentencia

```
media := (c1+c2)/2
```

al entrar en el procedimiento "sentencia" se llama al explorador. Este reconoce el identificador "media" y devuelve su puntero, id1, que queda guardado en una variable local de "sentencia". "sentencia" llama de nuevo al explorador, que devuelve ":", y luego a "expresión"; ésta llama a "término", y "término" llama al explorador y a "factor" (ver figuras 5.23 a 5.24), que, al encontrarse con "(", llama a su vez a "expresión". Dentro de esta ejecución de "expresión" se llama dos veces al explorador y a "término". Pues bien, las rutinas semánticas asociadas comprueban los tipos de c1 y c2 (cuyos punteros, id2 e id3, ha entregado el explorador) para ver si su suma es posible y construyen el cuarteto (apartado 5.2.1):

```
sum      id2      id3      id4
```

Se sale de este nivel de recursión en la ejecución de "expresión" al devolver el explorador el carácter ")". En este momento se ha construido, implícitamente, la parte más baja del árbol de la figura 5.13. Se completa luego la ejecución de la primera llamada a "expresión", en la que las rutinas semánticas construyen los dos siguientes cuartetos (apartado 5.2.1), y, al volver a "sentencia", la rutina semántica asociada a ésta construye el último cuarteto.

5.3 Intérpretes

La estructura de un programa intérprete es mucho más sencilla que la de un compilador. En esencia, puede representarse por el bucle de la figura 5.25. CONT_SENT es una variable que indica en cada momento la posición en el programa fuente de la siguiente sentencia a ejecutar. Su función es similar a la de "CONT_ENS" del ensamblador descrito en el apartado 5.1.3: en principio, se va incrementando en una unidad, pero puede tomar otro valor, dependiendo del análisis de la sentencia (si ésta es una llamada a procedimiento, o de con-

trol de un bucle, etc.). A cada tipo de sentencia posible se le asocia un procedimiento en el programa intérprete para ejecutar en la máquina real la acción especificada por la sentencia.

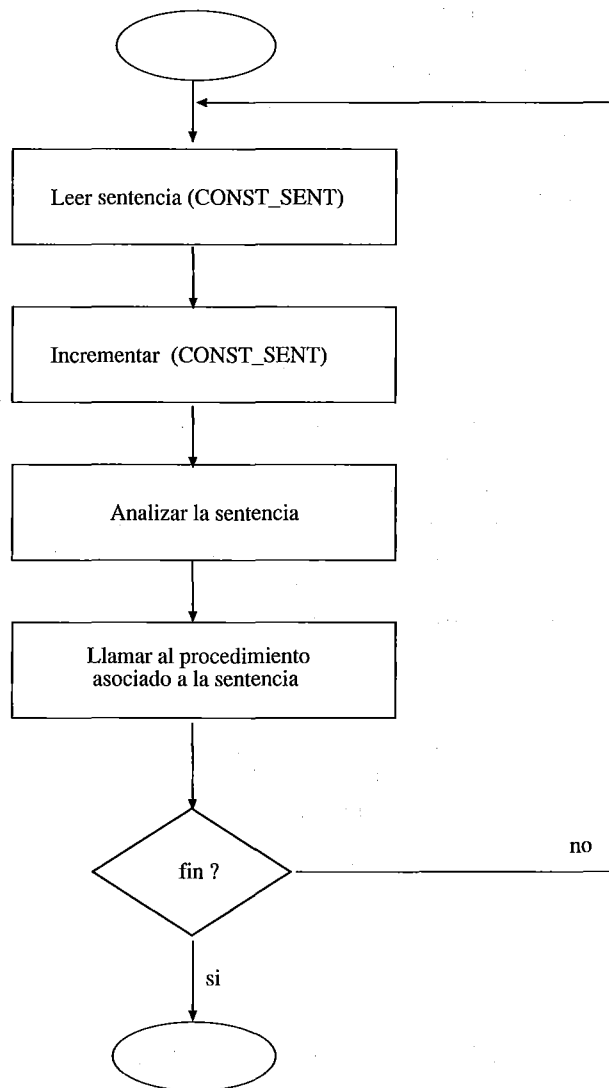


Figura 5.25.

El análisis de las sentencias puede estar guiado, como en los compiladores, por la definición formal de la sintaxis del lenguaje.

5.4 Procesadores de lenguajes declarativos

La sintaxis de un lenguaje funcional o lógico también puede formalizarse y expresarse con la notación BNF. De ahí que las fases de análisis léxico y sintáctico de los procesadores sean similares a las que hemos explicado para los procesadores de lenguajes procedimentales. El análisis semántico, la optimización y la generación de código son ya muy dependientes de cada lenguaje concreto.

Consideremos, por ejemplo, un intérprete para Prolog (apartado 2.4.3). Su funcionamiento puede basarse en el algoritmo de encadenamiento hacia atrás cuyo principio vimos en el apartado 2.4.5 del capítulo 6 del tema "Lógica". El seudocódigo que allí presentábamos deberá adaptarse, teniendo en cuenta las siguientes precisiones:

- El procedimiento "nodo_y" es el *procesador de objetivos* del intérprete de Prolog. La consulta del usuario es el objetivo inicial.
- El procedimiento "nodo_o" es el *procesador de reglas* (y de hechos: éstos no son más que reglas sin cuerpo).
- La "base de hechos" está formada por los hechos explícitamente declarados en el programa.
- Cuando no se encuentra solución, el procesador de reglas no pregunta al usuario; devuelve un "fracaso" (variable booleana "demostrado" con el valor "false") al procesador de objetivos, y si no hay más posibilidades, simplemente responde "NO"¹⁴. Como consecuencia, la base de hechos permanece inalterable durante la ejecución de una consulta.
- En el intérprete habrá que tener en cuenta mecanismos de sustitución y unificación que no se mencionaban en el seudocódigo (allí nos limitábamos al marco de la lógica de proposiciones, mientras que Prolog sigue el modelo de la lógica de predicados).
- El seudocódigo no especifica ninguna estrategia de control: el procedimiento "nodo_o" dice "seleccionar una regla", pero ¿con qué criterio se elige una cuando hay varias aplicables?; análogamente, "nodo_y" llama a "nodo_o" para cada uno de los objetivos pendientes, y no dice en qué orden se consideran esos objetivos. La estrategia más utilizada en los intérpretes de Prolog es la más sencilla: el procesador de reglas ("nodo_o") explora las reglas "de arriba abajo" (en el orden en que aparecen en el programa), y el procesador de objetivos ("nodo_y") llama al de reglas considerando los objetivos "de izquierda a derecha" (en el orden en que aparecen en el cuerpo de la regla que ha dado lugar a esos objetivos).

¹⁴ Es decir, el intérprete no funciona exactamente como lo haría el motor de inferencias de un sistema experto. Hay que decir, no obstante, que el lenguaje tiene muchas otras características no explicadas en el apartado 2.4.3, y que con ellas se pueden diseñar sistemas expertos con Prolog.

Para concretar el funcionamiento del intérprete de Prolog, aunque sea de manera básica e informal, consideremos el programa del apartado 2.4.3 y la consulta:

```
?-abuelo(X, luis) .
```

Esta consulta es el objetivo inicial. El intérprete tratará de refutar su negación, es decir, de encontrar algún valor para X tal que la cláusula "(abuelo(X, luis))" conduzca a una contradicción (cláusula vacía; quizás el lector debiera repasar ahora el apartado 4.8 del capítulo 4 del tema "Lógica", donde estudiábamos este mismo ejemplo, pero desde un punto de vista más formal y sin entrar en detalles procedimentales). Cuando se obtenga la refutación, en procedimiento "nodo" se pondrá la variable booleana "demostrado" con el valor "true".

Ante este objetivo inicial, el procesador de objetivos llama al procesador de reglas, que empieza desde la primera cláusula del programa ("madre(ana, luis)") tratando de encontrar una refutación.

Evidentemente, no la encuentra ("abuelo" \neq "madre") y tampoco con el siguiente hecho. La regla que le sigue (tercera cláusula), escrita en forma clausulada, es:

$$\text{padre}(X, Y) \vee \neg (\text{padre}(Y, Z) \vee \text{abuelo}(X, Z))$$

Entre esta cláusula y el objetivo negado es aplicable la regla de resolución, con la sustitución {luis/Z}, resultando " $\neg \text{padre}(X, Y) \vee \neg \text{padre}(Y, \text{luis})$ ", es decir, dos subobjetivos. El procesador de objetivos se fija en el primero y, con él como argumento, llama al procesador de reglas. Éste vuelve a explorar el programa desde arriba, y encuentra una refutación con la segunda cláusula, con la sustitución {josé/X, ana/Y}, sustitución que devuelve al procesador de objetivos. A éste le queda ahora otro: " $\neg \text{padre}(\text{ana}, Z)$ ". Obsérvese que el procesador no "sabe" más que lo que hemos declarado en nuestro programa Prolog: como no se lo hemos dicho, no puede saber que "ana" no puede ser "padre" de nadie. Por tanto, vuelve a llamar al procesador de reglas, que, obviamente, no puede refutarlo, devolviendo la variable "demostrado" con el valor "false".

Pero no termina aquí la búsqueda. En efecto, la primera vez que entró en funcionamiento, el procesador de reglas había encontrado un posible camino para la solución a través de la tercera cláusula. Como esta vía ha fracasado, ensaya con la cláusula siguiente. Creemos que el lector puede ya reproducir la nueva secuencia de acontecimientos y ver que se obtiene finalmente una refutación al objetivo inicial, con la sustitución {josé/X, luis/Y, ana/Z}. Como la consulta incluía la variable X, el intérprete responde, finalmente:

```
X = josé.
```

Para concluir esta introducción al procesamiento de Prolog, haremos algunas observaciones:

- a) La dificultad que quizás tenga el lector para seguir la descripción anterior radica en la naturaleza recursiva de los algoritmos. Si dispone de alguna experiencia en la programación con procedimientos recursivos en lenguajes tradicionales (imperativos) lo entenderá mucho mejor.
- b) Más dificultoso aún es el seguimiento del proceso cuando el propio programa Prolog contiene reglas recursivas. Sugerimos al lector que lo haga con la definición de "antepasado" del apartado 2.4.3 (la tarea resulta mucho más llevadera apoyándose en alguna construcción gráfica, como un árbol "Y-O" similar al de la figura 6.1 del tema "Lógica").
- c) Comprendido el principio, el lector admitirá que es igualmente aplicable a programas que tengan cientos o miles de hechos y/o reglas. El conjunto de hechos puede verse como una base de datos relacional. Al añadir las reglas se convierte en una *base de datos deductiva*.¹⁵

5.5 Herramientas para la generación de procesadores de lenguaje

No es difícil comprender que el diseño y la programación de un procesador es una tarea ardua (aunque normalmente se programe en un lenguaje de alto nivel, para el cual, por supuesto, tiene que existir previamente un compilador). Por otra parte, como hemos esbozado en los apartados anteriores, la tarea es bastante sistemática, especialmente en lo que se refiere a los analizadores léxico y sintáctico. No es, por tanto, de extrañar que hace tiempo surgiera la idea de automatizar esta tarea. Es decir, para el caso de los compiladores, diseñar un programa que acepte como datos de entrada las descripciones (en un metalenguaje, como BNF) de tres lenguajes:

- el lenguaje fuente, LF, en el que se escriben los programas que ha de traducir el compilador,
- el lenguaje objeto, LO, correspondiente a los programas traducidos por el compilador, y
- el lenguaje de programación, LP, en el que ha de estar escrito el propio compilador,

¹⁵ Pero en las bases de datos deductivas, al residir los hechos en memoria secundaria, es imposible, por razones de eficiencia, aplicar el algoritmo que hemos descrito. Remitimos al lector interesado a un libro ya referenciado en el tema "Lógica", el de Ullman (1988, 1989).

y que genere como salida un programa compilador (escrito en LP) tal que cuando se ejecute (después de haber sido traducido con un compilador de LP) acepte programas escritos en LF y los traduzca a LO. Tal programa se llama *compilador de compiladores*.¹⁶

Así expresada, la idea es demasiado ambiciosa y general. Actualmente se simplifica del siguiente modo:

- LP es fijo, es decir, el compilador de compiladores está diseñado para generar compiladores escritos en algún lenguaje predeterminado (C, Pascal y Ada son los más comunes).
- En lugar de un compilador de compiladores totalmente integrado se dispone de herramientas separadas para distintas fases de la compilación

Las herramientas de este tipo más conocidas son "Lex" y "Yacc", diseñadas para el sistema operativo UNIX, y que utilizan el lenguaje C como LP. Lex es un generador de exploradores: a partir de la definición de una gramática regular dada por una expresión regular genera el correspondiente explorador. Yacc ("Yet another compiler-compiler") es un generador de analizadores sintácticos: a partir de una gramática libre de contexto, dada por las reglas BNF, genera un analizador que sigue un algoritmo de reconocimiento ascendente. Aunque son programas separados, Lex y Yacc están diseñados para funcionar conjuntamente, de acuerdo con el esquema de principio que veíamos en la figura 5.12.

6. Resumen

Los lenguajes de *bajo nivel* (ensambladores y macroensambladores) son cercanos a los lenguajes de las máquinas reales, mientras que los de *alto nivel* facilitan la tarea de codificar algoritmos, porque independizan al programador de los detalles concretos del hardware y le permiten concentrarse en el problema a resolver.

En cualquier caso, siempre se puede considerar que todo lenguaje define a una máquina que lo "entiende", es decir, que reconoce los programas escritos en ese lenguaje y los ejecuta. Si el lenguaje no corresponde al de una máquina "real" (construida con hardware solamente), entonces la máquina definida es una *máquina virtual*.

¹⁶ Seguramente el lector tendrá que releer varias veces este párrafo para captar completamente la idea. La dificultad radica en que ahora tenemos varios niveles de lenguajes: las entradas al compilador de compiladores están escritas en un metalenguaje que describe a LF, LO y LP, y las entradas al compilador generado son programas escritos en LF. Obsérvese también que si se considera que un compilador es una "herramienta", entonces un compilador de compiladores es una "metaherramienta" (una herramienta que ayuda a crear herramientas).

Los *procesadores de lenguaje* son programas que permiten ejecutar en una determinada máquina (real o virtual), MO, programas escritos para otra máquina (normalmente virtual), MF. El lenguaje de MF se llama *lenguaje fuente*, y el de MO, *lenguaje objeto*. Hay dos tipos esencialmente diferentes de procesadores: los *traductores* y los *intérpretes*. Los primeros toman como datos la sucesión de caracteres que constituyen un programa fuente y generan un programa objeto que posteriormente se puede ejecutar en MO. Los traductores para lenguajes ensambladores se llaman *ensambladores*, y los diseñados para lenguajes de alto nivel, *compiladores*. Los intérpretes no generan de una sola vez todo el programa objeto correspondiente a un programa fuente, sino que van analizando sucesivamente las sentencias de éste y generando sobre la marcha las instrucciones de la MO necesarias para ejecutar esas sentencias.

Los lenguajes tradicionales, sean de bajo o alto nivel, suelen ser *imperativos*: las instrucciones o sentencias expresan órdenes elementales para una máquina, real o virtual. La idea de los lenguajes *declarativos* es que el programa no sea una sucesión de órdenes, sino una especificación del problema. En los lenguajes de *programación funcional* esta idea se plasma en tratar de expresar los problemas a través de aplicaciones de funciones matemáticas, mientras que en los lenguajes de *programación lógica* se trata de formular los problemas por medio de sentencias de la lógica formal.

La *definición formal de la sintaxis* de un lenguaje de programación, sea mediante la notación BNF o mediante diagramas sintácticos, es de gran utilidad en el diseño de las fases de análisis léxico y sintáctico de los procesadores de lenguaje. Generalmente, la parte que define las *categorías sintácticas* del lenguaje (palabras clave, identificadores, números, etc.) a partir de los caracteres elementales (símbolos terminales) se puede definir mediante una *gramática regular*, a la que corresponde un reconocedor finito, que se materializa en el programa *analizador léxico* o *explorador*. El resto de la definición sintáctica se formaliza como una *gramática libre de contexto*, a la que corresponde un reconocedor de pila, y que da lugar al programa *analizador sintáctico*. La función del explorador es leer caracteres del programa fuente de entrada y reconocer las distintas categorías sintácticas, mientras que la del analizador sintáctico consiste básicamente en reconstruir el árbol de derivación del programa en cuestión según la gramática del lenguaje. Las *rutinas semánticas* se ocupan de la traducción, sea al lenguaje objeto final, o a un código intermedio.

La *definición formal de la semántica* es un asunto más difícil, y existen actualmente varios enfoques (*operacional*, *denotacional* y *axiomático*) que pueden aplicarse al diseño de nuevos lenguajes, a la verificación y construcción sistemática de programas y al diseño de herramientas y entornos de programación potentes.

Un *compilador de compiladores* es un programa que genera un compilador a partir de las definiciones formales de los lenguajes de entrada (lenguaje fuente) y salida (lenguaje objeto).

7. Notas histórica y bibliográfica

La historia de los lenguajes de programación es paralela a la de los modelos de máquinas computadoras, reales o virtuales. Suele decirse que el primer programador fue Ada Augusta, Condesa de Lovelace, que a mediados del siglo pasado propuso diversos programas para una máquina que jamás llegó a construirse, el "analytical engine" de Babagge.

Pero el comienzo de la historia moderna de los ordenadores está jalonado por el modelo de von Neumann, propuesto en 1946, y la aparición del primer ordenador comercial, el UNIVAC I, en 1951. En esta época aún se programaba en lenguaje de máquina. En 1952, en el M.I.T., se diseñó un lenguaje llamado SAP (Symbolic Assembly Program), un ensamblador para el IBM 701. Y en el mismo año, Grace Murray Hopper publicaba el primer artículo sobre compiladores, en el que se describía "A-O", un compilador para el UNIVAC I. Por entonces, estas investigaciones se consideraban dentro del campo de la inteligencia artificial llamado "programación automática". En Knuth y Pardo (1980) se dan muchos detalles de la historia de los primeros tiempos de la programación.

Al final de los años 50 aparecen ya las primeras versiones de tres lenguajes de alto nivel que todavía siguen siendo ampliamente utilizados: FORTRAN (Backus *et al.*, 1957), COBOL (cuyo primer borrador data de 1959) y LISP (McCarthy, 1960). Backus propuso su notación en 1959, para describir el "International Algebraic Language" (Backus, 1959), precursor de Algol (Naur, 1963). Y heredero de Algol es Pascal, diseñado por Niklaus Wirth (1971), y que, a su vez, ha sido inspirador de otros lenguajes más modernos, como Ada y Modula-2. En la primera edición del manual de Pascal se introducían los diagramas sintácticos como alternativa a la notación BNF. Como ya hemos dicho en el apartado 2.4.2, la programación funcional tiene su origen teórico en el cálculo lambda, propuesto por el matemático Alonzo Church (1936). Un artículo ya clásico sobre los lenguajes aplicativos es el de Backus (1978). En cuanto a Prolog, procede de los trabajos de un equipo de investigadores de Marsella sobre lenguaje natural (Colmerauer *et al.*, 1973). Cabe citar también una comunicación de Kowalski (1974) en la que por primera vez se expresan claramente las ideas básicas de la programación lógica.

La semántica operacional está ligada a los estudios sobre relaciones entre lenguajes y máquinas intérpretes, como los de McCarthy y Painter (1967) y Allen *et al.* (1972). El origen de la semántica denotacional se encuentra en la "teoría de dominios" de Scott y Strachey (1971) y en la "teoría del punto fijo" de Manna (1974). La semántica axiomática está relacionada con la preocupación por demostrar matemáticamente la corrección de los programas. Un precursor en esta línea es el creador de LISP, John McCarthy: "en lugar de comprobar los programas con casos de prueba hasta que están depurados, deberíamos poder demostrar que tienen las propiedades deseadas" (McCarthy,

1962). La teoría fue desarrollada por diversos autores, principalmente Floyd (1967), Hoare (1969) y Dijkstra (1975). El transformador de predicados que aquí hemos llamado "pmd" lo introdujo Dijkstra con el nombre de "wp" ("weakest precondition").

Para ampliar los contenidos de este capítulo recomendamos los siguientes libros:

Sobre lenguajes de programación en general, Horowitz (1984) describe características comunes (tipos, procedimientos, abstracciones de datos, etc.) y dedica tres capítulos a la programación funcional y a los lenguajes de flujo de datos y orientados a objetos. El libro de Tucker (1986) tiene otro enfoque: describe, en sucesivos capítulos, las características esenciales y específicas de los lenguajes más conocidos. En una línea más conceptual, es muy interesante la lectura de un artículo de Ambler *et al.* (1992), que contiene una revisión y comparación de los distintos paradigmas de programación.

De Pascal, la referencia básica es el manual de Jensen y Wirth (1985). Textos didácticos sobre el lenguaje hay muchos. En español, además de los ya citados en el capítulo 4 del tema "Algoritmos", recomendamos el de Cueva *et al.* (1994), que no se limita al lenguaje: es una introducción a la programación estructurada y a la programación orientada a objetos.

Sobre programación funcional, Bird y Wadler (1988) es un libro introductorio y general, y Wikstrom (1987) está centrado en el lenguaje ML. Para LISP en particular, el texto más conocido es el de Winston y Horn (1991).

La referencia básica de Prolog (con la llamada "sintaxis de Edimburgo", que es la más seguida, y la que hemos utilizado en la breve descripción del apartado 2.4.3) es el libro de Clocksin y Mellish (1984). También cabe recomendar los de Lloyd (1987) y Sterling y Shapiro (1986). Los de Walker *et al.* (1987) y Bratko (1990) son especialmente interesantes, porque combinan Prolog con aplicaciones a la inteligencia artificial (el inconveniente del primero es que no utiliza la sintaxis de Edimburgo, sino la de un producto comercial de IBM). El libro de Maier y Warren (1988) presenta de manera muy clara y detallada tanto los aspectos funcionales (definición y uso del lenguaje) como los procesales (funcionamiento de los procesadores).

Una introducción a la programación funcional (con aplicaciones a la definición de lenguajes de programación) y la programación lógica (con los fundamentos de lógica formal) puede encontrarse en Delgado y Fernández (1994).

Sobre procesadores de lenguajes, Aho *et al.* (1986) sigue siendo el mejor libro de texto. Más básico y centrado en los fundamentos es el de Pittman y Peters (1992) (y, también, al ser más reciente, incluye temas más actuales; por ejemplo, técnicas de optimización específicas para procesadores RISC y vectoriales). Lex y Yacc se describen con todo detalle en Levine *et al.* (1992). Para la descripción del algoritmo de análisis sintáctico en el apartado 5.2.3 nos hemos basado en Wirth (1976), un magnífico libro (y con muy buena traducción) sobre programación y estructuras de datos que, pese a su antigüedad, sigue siendo recomendable.

Para un estudio completo de la definición formal de lenguajes y de su procesamiento, con notación y terminología unificadas, pueden seguirse los tres libros de Watt (1990: "conceptos y paradigmas", 1991: "syntaxis y semántica", 1992: "procesadores").

Por último, tenemos que citar el libro de Davis (1994), no sólo porque explica más detalladamente de lo que hemos podido hacer aquí la semántica, sino, sobre todo, porque cubre los mismos temas que este libro (lógica, autómatas, algoritmos y lenguajes) en un nivel algo más avanzado de formalización y conceptualización.

Referencias bibliográficas

- Aho, A.V., Sethi, R. y Ullman, J.D. *Compilers. Principles, techniques, and tools*. Addison-Wesley, Reading, Mass., 1986.
- Aikins, J.S. Prototypical knowledge for expert systems. *Artificial Intelligence*, 20 (1983), pp. 163-210.
- Alabau, A. y Figueras, J. *Algoritmos y máquinas*. Universidad Politécnica de Barcelona, C.P.D.A., 1.975.
- Alagic, S. y Arbib, M.A. *The design of well-structured and correct programs*. Springer-Verlag, N.Y., 1978.
- Alfonseca, M. Frames, semantics networks and object-oriented programming in APL2. *IBM J. Res. Dev.*, 33:5, pp. 502-510, Sep. 1989.
- Alfonseca, M. y Alcalá, A. *Programación orientada a objetos*. Anaya Multimedia, Madrid, 1992.
- Allen, C.D., Chapman, D.N. y Jones, C.B. *A formal definition of ALGOL 60*. IBM, TR. 12.105, Hursley (UK), ago. 1972.
- Allen, J.F. An internal-based representation of temporal knowledge. *Proc. 7th Int. Joint Conf. Artif. Int.*, Vancouver, Canada, 1981, pp. 221-226.
- Ambler, A.L., Burnett, M.M. y Zimmerman, B.A. Operational versus definitional: a perspective on programming paradigms. *Computer*, 25, 9 (Sep. 1992), pp. 28-43.

- Anderson, J.A. y Rosenfeld, E. (eds.). *Neurocomputing: foundations of research*. MIT Press, Cambridge, Mass., 1988.
- Arbib, M.A. *Brains, machines and mathematics*. McGraw-Hill paperbacks, 1965. (Existe versión castellana en la colección Alianza Universidad).
- Arbib, M.A. *Theories of abstract automata*. Prentice-Hall, Englewood Cliffs, N.J., 1969.
- Arbib, M.A. *Computers and the cybernetic society*. Academic Press, 1977. (Publicado en castellano por Editorial AC, bajo la supervisión de F. Sáez Vacas, 1978).
- Ashby, W.R. *An introduction to cybernetics*. Chapman and Hall, Londres, 1956. (Traducción de J. Santos: *Introducción a la cibernética*. Nueva Visión, Buenos Aires, 1960).
- Backus, J.W. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21, 8 (1978), pp. 613-641.
- Backus, J.W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *Proc. Internat. Conf. Information Processing*, UNESCO, París, 1959 (Butterworth's, Londres, 1960, pp. 125-132).
- Backus, J.W., Beeber, R.J., Best, S., Goldberg, R., Haibt, L.M., Herrick, H.L., Nelson, R.A., Sayre, D., Sheridan, P.B., Stern, H., Ziller, I., Hughes, R.A. y Nutt, R. The FORTRAN automatic coding system. *Proc. Western Joint Computer Conf.*, AIEE (actualmente IEEE), Los Angeles, 1957.
- Bar-Hillel, Y., Perles, M. y Shamir, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik Sprachwissenschaft und Kommunikationsforschung*, 14 (1961), 143- 172.
- Bartee, T.C. *Digital computers fundamentals*. McGraw-Hill, N.Y., 1960 (sexta ed.: 1985).
- Bartee, T.C. *Computer architecture and logic design*. McGraw-Hill, N.Y., 1991.
- Bartee, T.C., Lebow, I.L. y Reed, I.S. *Theory and design of digital machines*. McGraw-Hill, N.Y., 1962.
- Barwise, J. y Etchemendy, J. *Turing machines*. Accesible en la Internet en la dirección: <http://csli-www.stanford.edu/hp/Turing1.htm>
- Beck, K. y Cunningham, W. A laboratory for teaching object-oriented thinking. *Proc. Object-Oriented Programming Systems, Languages and Applications 1989 (OOPSLA'89)*. SIGPLAN Notices, Vol. 24, No 10, october 89, pp. 1-6.
- Bellew, R.K. y Booker, L.G. (eds.) *Proceedings of the fourth international conference on genetic algorithms and their applications*. Morgan Kaufmann, San Mateo, Calif, 1991.

- Bird, R. y Wadler, P. *Introduction to functional programming*. Prentice-Hall, Englewood Cliffs, N.J., 1988.
- Bochmann, G.V. *Architecture of distributed computer systems*. Springer-Verlag, Berlin, 1979.
- Bochvar, D. On three-valued logical calculus and its application to the analysis of contradictions. *Matematicheskij Sbornik*, 4 (1939), pp. 353-369.
- Böhm, C. y Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM*, 9, 5, may. 1966.
- Booch, G. *Object-oriented analysis and design with applications*. Benjamin/Cummings Publishing Company, Redwood City, Cal., 1994.
- Boole, G. *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities*. McMillan, London, 1854. Reimpr: Dover Publ. Inc., N.Y., 1958.
- Booth, T.L. *Sequential machines and automata theory*. Wiley, N.Y., 1967.
- Bratko, I. *Prolog programming for artificial intelligence* (2nd. ed.) Addison-Wesley, Reading, Mass., 1990.
- Brownston, L., Farrel, R., Kant, E. y Martin, N. *Programming expert systems in OPS5. An introduction*. Addison-Wesley, Reading, Mass., 1985.
- Brozozowski, J.A. A survey of regular expressions and their applications. *IRE Trans. Elec. Comp.*, EC-11 (1962), pp. 324-325.
- Brozozowski, J.A. Regular expressions for lineal sequential circuits. *IEEE Trans. Elec. Comp.*, EC-14 (1965), pp.148-156.
- Buchanan, B.G. y Shortliffe, E.H. (eds.) *Rule-based expert systems*. Addison-Wesley, Reading, Mass., 1984.
- Buchanan, B.G., Sutherland, G.L. y Feigenbaum, E.A. Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry. En B. Meltzer y D. Michie (Eds.): *Machine Intelligence*, vol 4, Edinburgh Univ. Press, 1969, pp. 209-254.
- Cantor, D.C. On the ambiguity problem of Backus systems. *Journal of ACM*, 9 (1962), 4, 477-479.
- Cao, T. y Sanderson, A.C. Task sequence planning using fuzzy Petri nets. *IEEE Trans. Sys., Man, and SMC*25, 5 (May 1995), pp. 755-768.
- Carbonell, J. (ed.) *Machine learning. Paradigms and methods*. Elsevier, Amsterdam, 1989.
- Carbonell, J.G., Cullingford, R. y Gershman, A. Steps towards knowledge-based machine translation. *IEEE Trans. Pattern Anal. and Mach. Int.*, 3, 4 (Jul. 1981), pp. 376-392.
- Cerrada, J.A. y Collado, M. *Programación I*. Universidad Nacional de Educación a Distancia, 1993.

- Chand, S. y Chiu, S.L. (Guest eds.) Special issue on fuzzy logic with engineering applications. *Proc. IEEE*, 83, 3 (Mar. 1995), pp. 343-483 (incluye 7 contribuciones de 22 autores).
- Chandrasekaran, B., Mittal, S. y Smith, J.N. RADEX. Towards a computer-based radiology consultant. En E.S. Gelsema y L.N. Kanal (eds.): *Pattern recognition in practice*. North-Holland, Amsterdam, 1980, pp. 463-474.
- Charniak, E. y McDermott, D. *Introduction to artificial intelligence*. Addison-Wesley, Reading, Mass., 1985.
- Chomsky, N. Three models for the description of language. *I.R.E. Trans. Inf. Theory*, IT-2 (1956), 113-114.
- Chomsky, N. On certain formal properties of grammars. *Information and Control*, 2 (1959), 137-167.
- Chomsky, N. *Context-free grammars and pushdown storage*. Quart. Prog. Dept. No. 65 (1962), MIT Res. Lab. Elect., 187-194.
- Chomsky, N. Prólogo al libro de Gross y Lentin (1967).
- Chomsky, N. *Language and mind*. Harcourt Braze, N.Y., 1968.
- Chomsky, N. *The logical structure of linguistic theory*. Plenum Press, N.Y., 1975.
- Chomsky, N. y Miller, G.A. Finite state languages. *Information and Control*, 1 (1958), 91-112.
- Chomsky, N. y Schutzenberger, M.P. *The algebraic theory of context-free languages. Computer Programming and Formal Systems*. North-Holland, Amsterdam, 1963, 118-161.
- Church, A. An unsolvable problem in elementary number theory. *American Journal of Mathematics*, 58 (1936), pp. 345-363.
- Clocksin, W.F., Darlington, J., Kennaway, S.R. y Sleep, M.R. "Part II. Declarative systems". En Chambers, F.B., Duce, D.A. y Jones, G.P. (eds.): *Distributed computing*. Academic Press, Londres, 1984, pp. 55-138.
- Cloksin, W.F. y Mellish, C.S. *Programming in Prolog*. Springer Verlag, Berlín, 1981.
- Coad, P. y Yourdon, E. *Object-oriented analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- Coad, P. y Yourdon, E. *Object-oriented design*. Prentice-Hall, Englewoods Cliffs, NJ, 1991.
- Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM*, 13, 6 (1970), pp. 337-387.
- Cohen, D. *Introduction to computer theory*. John Wiley, N.Y., 1991.
- Colmerauer, A., Kanoni, H., Pasero, R. y Roussel, P. *Un système de communication homme-machine en français*. Groupe d'Intelligence Artificielle. Université d'Aix-Marseille. Luminy, 1973

- Corge, Ch. *Eléments d'informatique*. Larousse Université, París, 1975.
- Coulter, N.S. Software science and cognitive psychology, *IEEE Trans. Softw. Eng.*, SE-9, 2 (mar. 1983), pp. 166-171.
- Cox, B. *Object-oriented programming: an evolutionary approach*. Addison-Wesley, 1986.
- Cuena, J. *Lógica informática*. Alianza, Madrid, 1986.
- Cuena, J., Fernández, G., Verdejo, F. y López de Mántaras, R. *Inteligencia artificial: sistemas expertos*. Alianza, Madrid, 1986.
- Cueva, J.M., García, P.A., López, B., Luengo, M.C. y Alonso, M. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Cuaderno Didáctico núm. 69, Dep. Matemáticas, Universidad de Oviedo. Gráficas Oviedo, 1994.
- Curtis, M.W. A Turing machine simulator. *Journal of the ACM*, 12, 1, (ene. 1965), pp. 1-13.
- De Morgan, A. *Formal logic*. Londres, 1847.
- Dahl, O. y Nygaard, K. SIMULA: an Algol-based simulation language. *Comm. ACM*, Vol. 9, No. 9, pp. 671-678, sept. 1966.
- Dahl, O.I., Myhrhaug, B. y Nygaard, K. *The SIMULA-67 common base language*. Rept. S-22, Norwegian Computing Centre, Oslo, 1970.
- Dahl, O.I., Dijkstra, E.W. y Hoare, C.A.R. *Notes on structured programming*. Academic Press, 1972.
- Datamation. Revolution in programming. *Datamation*, Dic. 1973. Número dedicado preferentemente al tema de la programación estructurada.
- Davis, M. y Putnam, H. A computing procedure for quantification theory. *Journal of the ACM*, 7, 3 (jul. 1960), pp. 201-215.
- Davis, M.D., Sigal, R. y Weyuker, J. *Computability, complexity and languages: fundamentals of computer science* (2nd. ed.) Academic Press, Sand Diego, Calif., 1994.
- Deaño, A. *Introducción a la lógica formal*. Alianza, Madrid, 1974. (3a. ed., reimpr., 1986).
- DeJong, G. Capítulo 21 de Kodratoff y Michalski (1990).
- Delgado Kloos, C. *Simulador de Máquina de Turing*. Proyecto Fin de Carrera, E.T.S.I. Telecomunicación, Madrid, Sep. 1978.
- Delgado Kloos, C. y Fernández, G. *Programación declarativa*. Servicio Publicaciones E.T.S.I.T.M., Madrid, 1994.
- Denning, P.J., Dennis, J.B. y Qualitz, J.E. *Machines, languages and computation*. Prentice-Hall, Englewood Cliffs, N.J., 1978.

- Díaz Cort, J., *Complejidad concreta y complejidad abstracta de algoritmos: Un panorama actual*. IV Escuela de Verano de Informática, Asociación Española de Informática y Automática, Granada, 1982.
- Díez Medrano, J., *Complejidad*. Trabajo para la asignatura de Fundamentos de Ordenadores I, Escuela Técnica Superior de Ingenieros de Telecomunicación, Madrid, junio 1984.
- Digitalk Inc. *Smalltalk/VP M tutorial and programming handbook*. Digitalk Inc., Los Angeles, Cal., 1989.
- Dijkstra, E.W. Guarded commands, non-determinacy and the formal derivation of programs. *Comm. ACM*, 18 (Aug. 1975), pp. 453-458.
- Dijkstra, E.W., *A discipline of programming*. Prentice-Hall, N.J., 1976.
- Driankov, D., Hellendoorn, H. y Reinfrank, M. *An introduction to fuzzy control*. Springer Verlag, Berlin, 1993.
- Dubois, D. y Prade, H. *Fuzzy sets and systems. Theory and applications*. Academic Press, New York, 1980.
- Dubois, P. y Prade, H. *Théorie des possibilités. Application à la représentation des connaissances en informatique*. Masson, París, 1985.
- Dubois, D. y Prade, H. *Possibility theory. An approach to computerized processing of uncertainty*. Plenum Press, New York, 1988.
- Duda, R.O., Hart, P.E., Nilsson, N.J. y Sutherland, G. *Semantic network representation in rule-based inference systems*. En D.A. Waterman y F. Hayes-Roth (Eds.): *Pattern-directed inference systems*. Academic Press, N.Y., 1978, pp. 203-221.
- Durkin, J. *Expert systems catalog of applications*. Intelligent Computer Systems, Akron, Ohio, 1993.
- Evey, J. *The theory and application of pushdown store machines*. (Tesis Doctoral). Harvard University, Cambridge, Mass., 1963.
- Fayyad, U. y Uthurusamy, S. (eds.) *KDD-94: Proceedings of AAAI-94 knowledge discovery in databases workshop*. AAAI Press, 1994.
- Fernández, G. *Curso de ordenadores. Conceptos básicos de arquitectura y sistemas operativos*. Ed. Syserco, Madrid, 1994.
- Ferrater, J. y Leblanc, H. *Lógica matemática*. Fondo de Cultura Económica, México, 1955 (2° ed., 1962).
- Floyd, R.W. On ambiguity in phrase structure languages. *Comm. ACM*, 5 (1962), 10, 526-534.
- Floyd, R.W. Assigning meaning to programs. En J.T. Schwartz (ed.): *Mathematical aspects of computer science*. Proc. Symp. Amer. Math. Soc., 1967, pp. 19-33.

- Floyd, R.W. The syntax of programming languages-A survey, *I.R.E. Trans. Electronic Computers*, 14 1964, 4, 346-353.
- Fogel, L., Owens, A. y Walsh, M. *Artificial intelligence through simulated evolution*. John Wiley, New York, 1966.
- Frege, G. *Die Grundlagen der Arithmetik, eine logisch- mathematische Untersuchung über den Begriff der Zahl*. Breslau, 1884. Versión traducida al inglés en J. van Heijenoort (ed.): *From Frege to Gödel*. Harvard University Press, 1967.
- Frenkel, K. Piecing together complexity, and complexity and parallel processing: an interview with Richard Karp. *Comm. ACM*, 29, 2 (feb.1986), pp. 110-117.
- Friedberg, R.M. A learning machine: Part I. *IBM Journal of Research and Development*, 2, 1 (Jan. 1958).
- Friedberg, R.M. A learning machine: Part II. *IBM Journal of Research and Development*, 3, 3 (Jul. 1959).
- Frost, R.A. *Introduction to knowledge-based systems*. Collins, Londres, 1986.
- Gallant, S. Connexionist expert systems. *Comm. ACM*, 31 (1988), pp. 152-169.
- Gallant, S. *Neural networks learning and expert systems*. MIT Press, Cambridge, Mass., 1993.
- Gamella, M. (ed.) *La tecnología del software. Temática y situación en España*. Fundesco, Madrid, 1985.
- Gardner, M. Mathematical games: On cellular automata, self- reproduction, the garden of Eden, and the game of life. *Scientific American*, Feb., Mar., Apr. 1971; Jan. 1972.
- Garijo, M. et al. Tema 10: *Algoritmos y complejidad*. Apuntes complementarios para la asignatura Fundamentos de la Programación, Escuela Técnica Superior de Ingenieros de Telecomunicación, Madrid, 1986.
- Genesereth, M.R. y Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, Calif., 1987.
- Genesereth, M.R. *Computational logic. An introduction in twenty easy lessons* (1995). Accesible en la Internet en la dirección:
<http://logic.stanford.edu/cs157/index.html>
- Gilbert, W.J. *Modern algebra with applications*. John Wiley, N.Y., 1976.
- Gill, T. Laws of unreliability. *Datamation*, vol. 21, N.3, Mar. 1975.
- Ginsburg, S. *The mathematical theory of context-free languages*. McGraw-Hill, N.Y., 1966.
- Ginsburg, S. Lectures on context-free languages. En Arbib, M.A. (ed.): *Algebraic theory of machines, languages, and semigroups*. Academic Press, N.Y., 1968.

- Glorioso, R.M. y Colón, F.C. *Engineering intelligent systems*. Digital Press, Bedford, Mass., 1980.
- Gödel, K., Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I (sobre proposiciones formalmente indecidibles de los Principia Mathematica y sistemas relacionados). *Monatshefte für Mathematik und Physik*, 38, 1931, pp. 173-98.
- Goldberg, D.E. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, Mass., 1989.
- Goldberg, A. y Robson, D. *Smalltalk-80: the language and its implementation*. Addison-Wesley, Reading, Mass., 1983.
- Goldschlager, L. y Lister, A. *Computer science: a modern introduction*, Prentice-Hall, N.J., 1982.
- Gottfried, B.S. *Programación en C*. McGraw-Hill, Madrid, 1993.
- Grefenstette, J.J. *Proceedings of the first international conference on genetic algorithms and their applications*. Lawrence Erlbaum, Hillsdale, N.J., 1985.
- Grefenstette, J.J. *Proceedings of the second international conference on genetic algorithms and their applications*. Lawrence Erlbaum, Hillsdale, N.J., 1987.
- Gross, M. y Lentin, A. *Notions sur les grammaires formelles*. Gautier-Villars, París, 1967. (Traducción: Tecnos, Madrid, 1976).
- Gupta, M.M., Saridis, G.N. y Gaines, B.R. (eds.). *Fuzzy automata and decision processes*. North-Holland, Amsterdam, 1977.
- Gutowitz, H. *Cellular automata: theory and experiment*. MIT Press, Cambridge, Mass., 1991.
- Haak, S. *Deviant logic*. Cambridge University Press, 1974.
- Haak, S. *Philosophy of logics*. Cambridge University Press, 1978.
- Halstead, M.H. *Elements of software science*, Elsevier, N.Y., 1977.
- Harrison, M.A. *Introduction to switching and automata theory*. McGraw-Hill, N.Y., 1965.
- Harrison, M.A. *Introduction to formal language theory*. Addison Wesley, Reading, Mass. 1978.
- Hayes, P.J. In defense of logic. *Proc. 5th Int. Joint Conf. Artif. Intell.* (1977), pp. 559-565.
- Hayes, P.J. The logic of frames. En *The frame reader*. De Guyter, Berlín, 1979.
- Hehner, E.C. *The logic of programming*. Prentice-Hall, Englewood Cliffs, N.J., 1984.
- Hennie, F. *Introduction to computability*. Addison-Wesley, Reading, Mass., 1977.

- Hertz, J., Krogh, A. y Palmer, R.G. *Introduction to the theory of neural computation*. Addison-Wesley, Redwood, Calif., 1991.
- Hinton, G.E. y Sejnowski, T.J. *Learning and relearning in Boltzmann machines*. En Rumelhart y McClelland (1986), Vol.1, Ch.7.
- Hirst, G. *Anaphora in natural language understanding: a survey*. Springer-Verlag, Berlín, 1981.
- Hoare, C.A.R. An axiomatic approach to computer programming. *Comm. ACM*, 12 (Oct. 1969), pp. 576-580, 583.
- Holland, J.H. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. *Proc. 1959 Eastern Joint Comp. Conf.*, pp. 108-113.
- Holland, J.H. *Adaptation in natural and artificial systems*. Univ. Michigan Press, 1975.
- Holland, J.H. *Escaping brittleness: the possibilities of general-purpose learning algorithms applied to paralell rule-based systems*. En Michalski et al., 1986, pp.,593-623.
- Holland, J.H., Holyoak, K.J., Nisbett, R.E. y Thagard, P.R. *Induction. Processes of inference, learning, and discovery*. MIT Press, Cambridge, Mass., 1986.
- Hopcroft, J.E. y Ullman, J.D. *Formal languages and their relation to automata*. Addison-Wesley, Reading, Mass., 1969.
- Hopcroft, J.E. y Ullman, J.E. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Reading, Mass., 1979.
- Hopcroft, J.E. Máquinas de Turing. *Investigación y Ciencia*, 94, jul. 1984, pp. 8-19.
- Horgan, J. Claude E. Shannon. *IEEE Spectrum*, 29, 4 (Apr.1992), pp.72-75.
- Horowitz, E. *Fundamentals of programming languages*. Springer- Verlag, Berlin, 1984.
- Huffman, D.A. The syntesis of sequential switching circuits. *Journal Franklin Institute*, 257 (1954), 3, pp. 161-190, y 4, pp. 275-303.
- Hunt, E.B. *Artificial intelligence*. Academic Press, 1975.
- Hunt, E.B., Marin, J. y Stone, P.T. *Experiments in induction*. Academic Press, New York, 1966.
- Jackson, M.J. *Principles of program design*. Academic Press, Londres, 1975.
- Jacobson, I. *Object-oriented software engineering: a use case driven approach*. Addison-Wesley, Reading, Mass., 1992.
- Jensen, K. y Wirth, N. *Pascal user manual and report. ISO Pascal standard (3a. ed.)* Springer-Verlag, N.Y., 1985.

- Jones, C.F. *Systematic software development using VDM*. Prentice-Hall, Englewood Cliffs, N.J., 1986.
- Joyanes, L. *Programación en Turbo Pascal*. McGraw-Hill, Madrid, 1990.
- Kanal, L.N. y Lemmer, J.F. (eds.) *Uncertainty in artificial intelligence*. Elsevier, New York, 1986.
- Kandel, A. y Langholtz, G. (eds.) *Fuzzy control systems*. CRC Press, Boca Raton, 1994.
- Karnaugh, M. The map method for synthesis of combinational logic circuits. *Trans. AIEE*, 72, 9 (1953), pp. 593-599.
- Karp, R.M. Combinatorics, complexity, and randomness, *Comm. ACM*, 29, 2 (feb. 1986), pp. 98-109.
- Kelley, A. y Pohl, I. *Lenguaje C. Introducción a la programación*. Addison-Wesley Iberoamericana, 1987.
- Kelly, J. *Artificial intelligence. A modern myth*. Ellis Hordwood, Chichester, England, 1993.
- Kernighan, B. W. y Ritchie, D. M. *The C programming language*. Prentice-Hall, 1978. Existe 2a. edición y versión en castellano.
- Kleene, S. *Introduction to metamathematics*. North-Holland, Amsterdam, 1952. (Traducción de M. Garrido: *Introducción a la metamatemática*. Tecnos, Madrid, 1974).
- Kleene, S.C. *Representation of events in nerve nets and finite automata*. En *Automata studies*, Princeton Univ. Press, Princeton, N.J., 1956.
- Klir, G. y Folger, N. *Fuzzy sets, uncertainty, information*. Prentice-Hall, Englewood Cliffs, N.J., 1988.
- Knuth, D.E. y Pardo, L.T. *The early development of programming languages*. En N. Metropolis, J. Howlet y G.C. Rota (eds.): *A history of computing in the twentieth century*. Academic Press, N.Y., 1980, pp. 197-213.
- Knuth, D.E. Algoritmos. *Investigación y Ciencia*, num.9, jun. 1977, pp. 42-53.
- Kodratoff, Y. *Introduction to machine learning*. Pitman, Londres, 1988.
- Kodratoff, Y. y Michalski, R.S. (eds.) *Machine learning: An artificial intelligence approach, vol. III*. Morgan Kaufmann, San Mateo, Calif., 1990.
- Kohavi, Z. *Switching and finite automata theory*. McGraw-Hill, N.Y., 1970.
- Kosko, B. *Neural networks and fuzzy systems: a dynamical approach to machine intelligence*. Prentice-Hall, Englewood Cliffs, N.J., 1991.
- Kowalski, R. Predicate logic as a programming language. *Proc. IFIP 74*. North Holland, Amsterdam, 1974, pp. 569-574.
- Kowalski, R. *Logic for problem solving*. North-Holland, New York, 1979. (Traducción de J.A. Calle: *Lógica, programación e inteligencia artificial*. Díaz de Santos, Madrid, 1986).

- Kuroda, S.Y. Classes of languages and linear-bounded automata. *Information and control*, 7 (1964), 2, 114-125.
- Ladd, S. R. *C++ techniques and applications*. M & T Publishing, Redwood City, Cal., 1990.
- Langley, P., Zytkow, J.M., Simon, H. y Bradshaw, G.L. *The search for regularity: four aspects of scientific discovery*. En Michalski et al. (1986), pp.425-469.
- Lau, C. *Object-oriented programming using SOM and DSOM*. Van Nostrand Reinhold, New York, 1994.
- Lenat, D.B. Automated theory formation in mathematics. *Proc. 5th Int. Joint Conf. Artif. Intell.* (1977), pp.833-842.
- Lenat, D.B. *The role of heuristics in learning by discovery: three case studies*. En Michalski et al. (1983), pp.243-306.
- Levine, J.R., Mason, T. y Brown, D. *Lex&Yacc*, 2nd. ed.. O'Reilly and Associates, 1992.
- Lewis, C.I. y Langford, C.H. *Symbolic logic*. The Century Comp., N.Y., 1932 (2a. ed., Dover Publ., N.Y., 1959).
- Lewis, H.R. y Papadimitriou, C.H. *Elements of the theory of computation*. Prentice-Hall, Englewood Cliffs, N.J., 1988.
- Linger, R., Mills, H. y Witt, B. *Structured programming theory and practice*. Addison Wesley, Reading, Mass., 1979.
- Lloyd, J.W. *Foundations of logic programming*. Springer Verlag, New York, 1981.
- López de Mántaras, R. *Approximate reasoning models*. Ellis Harwood, Chichester, England, 1990.
- Love, T. *Object lessons*. SIGS Books, New York, 1993.
- Lukasiewicz, J. *On 3-valued logic*. (1920). Reproducido en McCall, S. (ed.): *Polish logic*. Oxford University Press, 1967.
- Lukasiewicz, J. *Many-valued systems of propositional logic*. (1930). Reproducido en McCall, S. (ed.): *Polish Logic*. Oxford University Press, 1967.
- Luria, A.R. *Cerebro y lenguaje*. Fontanella, Barcelona, 1974a.
- Luria, A.R. *Cerebro en acción*. Fontanella, Barcelona, 1974b.
- Madnick, S.E. y Donovan, J.J. *Operating systems*. McGraw-Hill, N.Y., 1974.
- Maier, D. y Warren, D.S. *Computing with logic. Logic programming with Prolog*. Benjamin-Cummings, Menlo Park, Calif., 1988.
- Mandado, E. *Sistemas electrónicos digitales*. (7a. ed.). Marcombo, Barcelona, 1991.

- Manna, Z. y Pnueli, A. The modal logic of programs. *Proc. 6th Int. Colloquium on Automata, Languages and Programming*. Springer Verlag (Lecture Notes in Computer Science, Vol. 71), 1979, pp. 385-411.
- Manna, Z. y Pnueli, A. *Verification of concurrent programs: the temporal framework*. En Boyer, R.S. y Moore, J.S. (eds.): *The correctness problem in computer science*. Academic Press, New York, 1981, pp. 215-273.
- Manna, Z., y Wolper, P. Synthesis of communicating processes form temporal logic. *ACM Trans. Progr. Lang. and Syst.*, 6 (1984), pp. 68-93.
- Manna, Z., *Mathematical theory of computation*. McGraw-Hill, N.Y., 1974.
- Mano, M.M. *Digital design*, 2nd. ed. Prentice-Hall, Englewood Cliffs, N.J., 1991.
- Mano, M.M. y Kime, C.R. *Logic and computer design fundamentals*. Prentice-Hall, Englewood Cliffs, N.J., 1994.
- Martin, J. y Odell, J. *Object-oriented analysis and design*. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- Martin, M.A. y Fateman, R.J. The MACSYMA system. *Proc. 2nd Symp. Symbolics and Algebraic Manipulation*. Los Angeles, Ca., 1971, pp. 59-75.
- Matz, D. *What Is a Turing machine simulator?* Accesible en la Internet en la dirección: <http://odin.wosc.osshe.edu/cs407/matzd/turing.html>
- McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.*, dic. 1976, pp. 308-320.
- McCarthy, J. y Painter, J. Correctness of a compiler for arithmetic expressions. En J.T. Schwartz (ed.): *Mathematical aspects of computer science*. American Mathematical Society, 1967, pp. 33-41.
- McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 3, 4 (1960), pp. 184- 195.
- McCarthy, J. A basis for a mathematical theory of computation. *Proc. IFIP Congress 62*, North Holland, Amsterdam, 1963.
- McCulloch, W.S. y Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5 (1943).
- McDermott, D. A temporal logic for reasoning about plans and actions. *Cognitive Science*, 6 (1982), pp. 101-155.
- McDermott, D. y Doyle, J. Non-monotonic logic. *Artificial Intelligence*, 13 (1980), pp. 27-39.
- McNaughton, R. y Yamada, H. Regular expressions and graphs for automata. *IRE Trans. Elec. Comp.*, EC-9 (1960), pp. 39-47.
- Mead, C. y Conway, L. *Introduction to VLSI systems*. Addison- Wesley, Reading, Mass., 1980.

- Mealy, G.H. A method for synthesising sequential circuits. *Bell System Tech. J.*, 34 (1955), pp. 1045-1079.
- Mendel, J.M. Fuzzy logic systems for engineering: a tutorial. *Proc. IEEE*, 83, 3 (Mar. 1995), pp. 345-377.
- Meyer, B. *Object-oriented software construction*. Prentice-Hall, Englewood Cliffs, NY, 1988.
- Michalewicz, Z. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag, Berlin, 1992.
- Michalski, R.S. *Variable-valued logic and its applications to pattern-recognition and machine learning*. En D. Rine (ed.): *Multiple-valued logic and computer science*. North-Holland, Amsterdam, 1975, pp. 506-534.
- Michalski, R.S., Carbonell, J.G. y Mitchell, T.M. (eds.) *Machine learning: An artificial intelligence approach*. Tioga, Calif., 1983. (Publicado en Europa por Springer Verlag, Berlin, 1984).
- Michalski, R.S., Carbonell, J.G. y Mitchell, T.M. (eds.) *Machine learning: An artificial intelligence approach*, vol. II. Morgan Kaufmann, Los Altos, Calif., 1986.
- Michalski, R.S. y Chilausky, R.L. Learning by being told and learning from examples: and experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy Analysis and Information Systems*, 4, 2 (Jun. 1980), pp. 125-160.
- Michalski, R.S. y Tecucci, G. (eds.) *Machine learning: a multistrategy approach*, vol. IV. Morgan Kaufmann, San Francisco, Calif., 1994.
- Mills, H.D. The new math of computer programming. *Comm. ACM*, 18, 1 (ene. 1975).
- Minski, M. A framework for representing knowledge. En P. Winston (ed.): *The psychology of computer vision*. McGraw-Hill, N.Y., 1975, pp. 211-277.
- Minsky, M. y Papert, S. *Perceptrons*. MIT Press, Cambridge, Mass., 1969.
- Mitchell, M., Crutchfield, J.P. y Hrabar, P.T. *Evolving cellular automata to perform computations: mechanisms and impediments*. Santa Fe Institute Working Paper 93-11-071. Accesible mediante "ftp" en la Internet:
servidor "santafe.edu"
directorio "/pub/Users/mm/"
ficheros "sfi-93-11-071.part1.ps.Z" y "sfi-93-11-071.part2.ps.Z".
- Mitra, S. y Pal, S.K. Fuzzy multilayer Perceptron, inferencing and rule generation. *IEEE Trans. Neural Networks*, 6, 1 (Jan. 1995), pp. 51-63.
- Mompin, J. (coord.) *Inteligencia artificial. Conceptos, técnicas y aplicaciones*. Marcombo, Barcelona, 1987.

- Moore, E.F. *Gedanken-experiments on sequential machines*. Automata studies: Annals of mathematical studies, No. 34, Princeton Univ. Press, Princeton, N.J., 1956, pp. 129-153.
- Moore, R.C. A formal theory of knowledge and action. En Hobbs, J.R. y Moore, R.C. (eds.): *Formal theories of the common sense world*. Ablex, Norwood, N.J., 1984.
- Morgan, Ch., ed., Smalltalk. *BYTE*, vol.6, no. 8, agosto 1981.
- Murata, T. Petri nets: properties, analysis, and applications. *Proc. IEEE*, 77, 4 (Apr. 1989), pp. 541-580.
- Myhill, J. *Linear bounded automata*. WADD Tech. Note 60-165, Wright Patterson Air Force Base, Ohio, 1960.
- Narendra, K.S. y Thathachar, M.A.L. *Learning automata. An introduction*. Prentice-Hall, Englewood Cliffs, N.J., 1989.
- Naur, P. (ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6, 1 (1963), pp. 1-17.
- Nilsson, J.J. *Principles of artificial intelligence*. Springer Verlag, Berlin, 1982. (Traducción de J. Fernández-Biarge: *Principios de inteligencia artificial*. Díaz de Santos, Madrid, 1987).
- Oberman, R.M.M. *Disciplines in combinational and sequential circuit design*. McGraw-Hill, N.Y., 1970.
- Oettinger, A.G. Automatic syntactic analysis and the pushdown store. *Proc. Symp. Applied Math., Amer. Math. Soc.*, Providence, Rhode Island, 1961.
- Osherson, D.N., Strob, M. y Weinstein, S. *Systems that learn. An introduction to learning theory for cognitive and computer scientists*. MIT Press, Cambridge, Mass., 1986.
- Ott, G. y Feinstein, N. Design of sequential machines from their regular expressions. *Journal of the ACM*, 8, 4 (1961), pp. 585-600.
- Patterson, D.A. y Hennessy, J.L. *Computer organization and design. The hardware/software interface*. Morgan Kaufmann, San Mateo, Calif., 1994. (Traducción de J.M. Sánchez: *Organización y diseño de computadores. La interfaz hardware/software*. McGraw-Hill Interamericana de España, Madrid, 1995).
- Pauker, S.G., Gorry, G.A., Kassirer, J.P. y Schwartz, W.B. Towards the simulation of clinical consultation. Taking a present illness by computer. *American Journal of Medicine*, 60 (1976), pp. 981-996.
- Pearl, J. *Probabilistic reasoning in intelligent systems*. Morgan Kaufmann, Palo Alto, Calif., 1988.
- Peterson, J.L. Petri nets. *Computing Surveys*, 9, 3 (sep. 1977), pp. 223-251.

- Petri, C.A. *Kommunikation mit automaten*. Univ. Bonn, 1962. (Versión traducida al inglés en Supplement 1 to Technical Report RADC-TR-65-377, vol. 1, Rome Air Development Center, Griffis Air Force Base, New York, 1965).
- Piatetsky-Shapiro, G. (ed.) *KDD-93: Proceedings of AAAI-93 knowledge discovery in databases workshop*. AAAI Press, 1993.
- Piatetsky-Shapiro, G. y Frawley, W.J. *Knowledge discovery in databases*. AAAI-MIT Press, Menlo Park, Calif., 1991.
- Pittman, T. y Peters, J. *The art of compiler design: theory and practice*. Prentice-Hall, Englewood Cliffs, N.J., 1992.
- Pople, H.R., Myers, J.D. y Miller, R.A. DIALOG: A model of diagnostic logic for internal medicine. *Proc. 4th Int. Joint Conf. Artif. Int.*. Tbilisi (URSS), 1975, pp. 848-855.
- Post, E. Formal reduction of the general combinatorial problem. *American Journal of Mathematics*, 65 (1943), pp. 197-268.
- Prerau, D.S. *Developing and managing expert systems. Proven techniques for business and industry*. Addison-Wesley, Reading, Mass., 1990.
- Pucknell, D.A. y Eshraghian, K. *Basic VLSI design*, 3rd. ed. Prentice-Hall, Englewood Cliffs, N.J., 1994.
- Pylyshyn, Z.W. *Perspectives on the computer revolution*. Prentice-Hall, Englewood Cliffs, N.J., 1970. (Traducción de L. García Llorente: *Perspectivas de la revolución de los computadores*. Alianza, Madrid, 1975).
- Pylyshyn, Z.W. y Bannon, L.J. *Perspectives on the computer revolution*, 2nd. ed. Ablex Publishing, Norwood, N.J., 1989.
- Quillian, M.R. *Semantic memory*. En M. Minsky (ed.): *Semantic information processing*. M.I.T. Press, Cambridge, Mass., 1968, pp. 354-402.
- Quinlan, J.R. *Discovering rules from large collections of examples: a case study*. En D. Michie (ed.): *Expert systems in the microelectronic age*. Edinburgh Univ. Press, 1979.
- Quinlan, J.R. Induction of decision trees. *Machine Learning*, 1 (1986), pp.81-106.
- Rabin, M.O. Probabilistic automata. *Information and control*, 6, 3 (1963), pp. 230-245.
- Rabin, M.O. y Scott, D. Finite automata and their decision problems. *IBM Journal Res. Dev.*, 3 (1959), 2, 114-125.
- Randell, B. y Russell, L.J. *Algol 60 implementation*. Academic Press, N.Y., 1964.
- Rawlings, G.J.E. (ed.) *Foundations of genetic algorithms*. Morgan Kaufmann, San Mateo, Calif., 1991.
- Reisig, W. *Petri nets*. Springer-Verlag, Berlin, 1985.
- Rescher, N. *Many-valued logic*. McGraw-Hill, New York, 1969.

- Ribeiro, J.L., Treleaven, P.C. y Alippi, C. Genetic-algorithm programming environments. *Computer*, 27, 6 (Jun. 1994), pp. 28-43.
- Rich, E. y Knight, K. *Artificial intelligence* (2nd. ed.) McGraw- Hill, New York, 1991. (Traducción de P.A. González y F. Trescastro: *Inteligencia artificial*. McGraw-Hill Iberoamericana de España, Madrid, 1994).
- Robinson, J.A. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 1 (ene. 1965), pp. 23-41.
- Robinson, J.A. *Logic: form and function*. Edinburg University Press, 1979.
- Rosenblatt, F. *Principles of neurodynamics, Perceptrons and the theory of brain mechanisms*. Spartan Books, Washington D.C., 1962.
- Rubin, K.S. y Goldberg, A. Object behavior analysis., *Comm. ACM*, vol. 35, no 9, pp. 48-62, sept., 1992.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. y Lorenson, W. *Object-oriented modelling and design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- Rumelhart, D.E. y McClelland, J.L. (eds.) *Parallel distributed processing*. MIT Press, Cambridge, Mass., 1986.
- Rumelhart, D.E., Hinton, G.E. y Williams, R.J. *Learning internal representations by error propagation*. En Rumelhart y McClelland (1986), Vol.1, Ch.8.
- Sáez Vacas, F. Guía para un análisis estructurado de la programación estructurada. Inforprim 1975. *Proceso de Datos*, num. 54, ene.-feb. 1976.
- Sáez Vacas, F. *Complejidad y tecnología de la información*. Instituto Tecnológico Bull, Madrid, 1992, reedición E.T.S.I. Telecom., Madrid, 1994.
- Salmon, W.I. *Introducción a la computación con Turbo Pascal*. Addison-Wesley Iberoamericana, 1993.
- Samuel, A. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3 (1959).
- Samuel, A. Some studies in machine learning using the game of checkers. Part II. *IBM Journal of Research and Development*, 11 (1967).
- Sanders, G. *Hardware design. A modern introduction*. Prentice- Hall, Englewood Cliffs, N.J., 1993.
- Scala, J.J. y Minguet, J.M. *Informática II. Unidad didáctica 3*. Universidad Nacional de Educación a Distancia, 1974.
- Schaffer, J.D. (ed.) *Proceedings of the third international conference on genetic algorithms and their applications*. Morgan Kaufmann, San Mateo, Calif, 1989.
- Schank, R.C. y Abelson, R.P. *Scripts, plans, goals, and understanding*. Lawrence Erlbaum, Hillsdale, N.J., 1977.
- Scott, D. y Strachey, C. Towards a mathematical semantics for computer languages. *Proc. Symp. Computers and Automata*, 1971, y en Technical Monograph PRG-6, Oxford Univ. Comp. Lab., pp. 19-46.

- Sejnowski, T.J. y Rosenberg, C.R. *NetTalk: a parallel network that learns to read aloud*. Technical Report TR-86-01, Dep. Electrical Eng., John Hopkins Univ., 1986.
- Shafer, G. *A mathematical theory of evidence*. Princeton University Press, Princeton, N.J., 1976.
- Shannon, C.E. The synthesis of two-terminal switching circuits. *Bell System Tech. J.*, vol. 28 (1949), pp. 59-98.
- Shannon, C.E. *A symbolic analysis of relay and switching circuits*. Van Nostrand, N.Y., 1954.
- Shastri, L. (guest ed.) Fuzzy logic symposium. *IEEE Expert*, 9, 4 (Aug. 1994), pp. 2-49 (incluye 17 contribuciones de 23 autores).
- Sheffer, H.M. A set of five independent postulates for boolean algebras, with application to logical constants. *Trans. Amer. Math. Soc.*, 14 (1913).
- Shields, M.W. *An introduction to automata theory*. Blackwell Scientific Publ., 1987.
- Shlaer, S. y Mellor, S.J. *Object-oriented system analysis: modelling the world in data*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
- Shooman, M.L. *Software engineering*. McGraw-Hill, International Student Edition, Auckland, 1983.
- Shortliffe, E.H. *Computer-based medical consultations*. Elsevier, N.Y., 1976.
- Silva, M. *Las redes de Petri en la automática y la informática*. AC, Madrid, 1985.
- Simpson, P.K. Fuzzy min-max neural networks - Part I: Classification. *IEEE Trans. Neural Networks*, 3, 5 (Sep. 1992), pp. 776-786.
- Simpson, P.K. Fuzzy min-max neural networks - Part II: Clustering. *IEEE Trans. Fuzzy Systems*, 1, 1 (Feb. 1993), pp. 32-45.
- Singh, J. *Teoría de la información, del lenguaje y de la cibernética* (2a. ed.) Alianza, Madrid, 1976.
- Skansholm, J. *Ada: from the beginning*. 2a. ed., Addison-Wesley, Wokingham, Engl., 1994.
- Skolem, T. *Über die mathematische logik*. *Norsk matematisk tidsskrift*, 10 (1928), pp. 125-142. Versión traducida al inglés en van Heijenoort, J. (ed.): *From Frege to Gödel*. Harvard University Press, 1967, pp.508-524.
- Soucek, B. *Neural and intelligent systems integration*. John Wiley, New York, 1991.
- Srinivas, M. y Patnaik, L.M. Genetic algorithms: a survey. *Computer*, 27, 6 (Jun. 1994), pp. 17-26.
- Steimann, F. y Adlassnig, K.P. Clinical monitoring with fuzzy automata. *Fuzzy sets and systems*, 61 (1994), pp. 37-42.

- Sterling, L. y Shapiro, E. *The art of Prolog*. MIT Press, Cambridge, Mass., 1986.
- Stewart, I. *Conceptos de matemática moderna*. Alianza, Madrid, 1977.
- Stone, H.S. *Introduction to computer organization and data structures*. McGraw-Hill, N.Y., 1972.
- Stroustrup, B. *The C++ programming language*. Addison-Wesley, Reading, Mass., 1986.
- Tabourier, Y., Rochfeld, A. y Frank, C. *La programmation structurée en informatique*. Les Editions d'Organisation, París, 1975.
- Tennant, H. *Natural language processing*. Petrocelli, New York, 1981.
- Tenny, T. Structured programming in Fortran. *Datamation*, jul. 1974.
- Tocci, R.L. *Digital systems* (6th ed.) Prentice-Hall, Englewood Cliffs, N.J., 1995.
- Toffoli, T y Margolus, N. *Cellular automata machines. A new environment for modeling*. MIT Press, Cambridge, Mass., 1987.
- Trakhtenbrot, B.A. *Algoritmos*. En Z.W. Pylyshyn: *Perspectivas de la revolución de los computadores*. Alianza Universidad, 1975, pp. 108-130.
- Trigoboff, M. y Kulikowski, C.A. IRIS: a system for the propagation of inferences in a semantic net. *Proc. 5th Int. Joint Conf. Artif. Intell.*, Cambridge, 1977, pp. 274-280.
- Tsetlin, M.L. Sobre el comportamiento de autómatas finitos en entornos aleatorios (en ruso). *Automatika i telemekhanika*, 22, 10 (oct. 1961), pp. 1345-1354.
- Tucker, A.B. *Programming languages* (2a. ed.) Mc-Graw-Hill, N.Y., 1986. (Traducción de J.M. Troya: *Lenguajes de programación*. McGraw-Hill, Madrid, 1987).
- Tucker, A.B., Bradley, W.J., Cupper, R.D., y Garnick, D.K. *Fundamentals of computing I*. McGraw-Hill (Schaum), Hightstown, NJ, 1992.
- Turing, A. Computing machinery and intelligence. Publicado originalmente en *Mind: a quaterly review of psychology and philosophy* (1950). Puede encontrarse traducción en Z.W. Pylyshyn: *Perspectivas de la revolución de los computadores*, Alianza, Madrid, 1975, pp. 305-333.
- Turner, R. *Logics for artificial intelligence*. Ellis Horwood, Chichester (UK), 1984.
- Ullman, J.D. *Principles of database and knowledge-base systems. Vol. I*. Computer Science Press, Rockville, Md., 1988.
- Ullman, J.D. *Principles of database and knowledge-base systems. Vol. II: The new technologies*. Computer Science Press, Rockville, Md., 1989.
- Varshavskii, V.I. y Vorontsova, I.P. Sobre el comportamiento de autómatas estocásticos con estructura variable (en ruso). *Automatika i telemekhanika*, 24, 3 (mar. 1963), pp. 353-360.

- Vega, M. de. *Introducción a la psicología cognitiva*. Alianza Editorial, Madrid, 1985.
- Veitch, E.W. A chart method for simplifying truth funtions. *Proc. ACM*, May. 1952, pp. 127-133.
- Virant, J. y Zimic, N. Fuzzy automata with fuzzy relief. *IEEE Trans. Fuzzy Systems*, 3, 1 (Feb. 1995), pp. 69-74.
- Von Neumann, J. *The general and logical theory of automata*. En *Cerebral machanisms in behaviour - The Hixon Symposium*. John Wiley, N.Y., 1951.
- Von Neumann, J. *Probabilistic logic and the syntesis of reliable organisms from unreliable components*. En C. Shannon y J. McCarthy (eds.): *Automata studies*, Princeton Univ. Press, Princeton, N.J., 1956.
- Von Neumann, J. (ed. A. Burks). *Theory of self-reproducing automata*. Univ. Illinois Press, Urbana, 1966.
- Wakerly, J.F. *Digital design*, 2nd. ed. Prentice-Hall, Englewood Cliffs, N.J., 1993.
- Walker, A., McCord, M., Sowa, J.F. y Wilson, W.G. *Knowledge systems and Prolog*. Addison-Wesley, Reading, Mass, 1987.
- Wang, H. Juegos, lógica y computadores. En *Computadoras y computación*. Ed. R.R. Fenichel y J. Weizenbaum. Blume, 1974, pp. 150-158. Edición inglesa: W.H. Freeman. (El artículo de Wang se publicó en 1965 en el Scientific American).
- Warnier, J.D. *Programación lógica. Tomos I y II*, Editores Técnicos Asociados, Barcelona, 1973.
- Watt, D.A. *Programming language concepts and paradigms*. Prentice-Hall, Engle-wood Cliffs, N.J., 1990.
- Watt, D.A. *Programming language syntax and semantics*. Prentice-Hall, Engle-wood Cliffs, N.J., 1991.
- Watt, D.A. *Programming language processors*. Prentice-Hall, Englewood Cliffs, N.J., 1993.
- Wee, W.G. y Fu, K.S. A formulation of fuzzy automata and its application as a model of learning systems. *IEEE Trans. Syst. Sci. Cyb.*, SSS5 (1969), pp. 215-223.
- Weiss, S.M. y Kulikowski, C.A. EXPERT: A system for developping consultations models. *Proc. 6th Int. Joint Conf. Artif. Int.*. Tokyo, 1979, pp. 942-947.
- Weiss, S.M., Kulikowski, C.A. y Safir, A. Glaucoma consultation by computer. *Computers in Biology and Medicine*, 8 (1978), pp. 25- 40.

- Whitehead, A.N. y Russel, B. *Principia Mathematica*. Cambridge, 1910 (vol. I), 1912 (vol. II), 1913 (vol. III). Segunda edición en Cambridge Univ. Press, 1950.
- Whitley, D. (ed.) *Foundations of genetic algorithms*. Morgan Kaufmann, San Mateo, Calif., 1992.
- Widrow, B. Generalization and information storage in networks of adaline "neurons". En M. Yovitz, G.T. Jacobi y G.D. Goldstein (eds.): *Self-Organizing Systems*. Spartan Books, Washington D.C., 1962, pp.435-461.
- Widrow, B. y Lehr, M.A. 30 years of adaptive neural networks: perceptron, madaline and backpropagation. *Proc. IEEE*, 78, 9 (Sep.1990), pp.1415-1442.
- Wilkinson, B. y Makki, R. *Digital system design*, 2nd. ed. Prentice-Hall, Englewood Cliffs, N.J., 1992.
- Winblad, A.L., Edwards, S.D. y King, D.R. *Object-oriented software*. Addison-Wesley, Reading, Mass., 1990.
- Winograd, S. y Cowan, J.D. *Reliable computation in the presence of noise*. M.I.T. Press, Cambridge, Mass., 1963.
- Winston, P.H. y Horn, B.K.P. *LISP* (3rd. ed.) Addison-Wesley, Reading, Mass., 1991.
- Wirfs-Brock, R., Wilkerson, B. y Wiener, L. *Designing object-oriented software*. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- Wirth, N. The programming language Pascal. *Acta Informática*, 1, 1 (1971), 35-36.
- Wirth, N. *Algorithms + data structures = programs*. Prentice-Hall, 1976 (Traducción de A. Alvarez y J. Cuenca: *Algoritmos + estructuras de datos = programas*. Ediciones del Castillo, Madrid, 1980, 4a. impr., 1985).
- Wirth, N. Algoritmos y estructuras de datos. *Investigación y Ciencia*, num. 98, nov., 1984, pp. 24-35.
- Wolf, W. *Modern VLSI design. A CAD-based approach*. Prentice-Hall, Englewood Cliffs, N.J., 1994.
- Wolfram, S. (ed.) *Theory and applications of cellular automata*. World Scientific, Singapore, 1986.
- Yager, R.R. y Filev, D.P. *Essentials of fuzzy modelling and control*. John Wiley, New York, 1994
- Yourdon, E., *Techniques of program structure and design*. Prentice-Hall, Englewood Cliffs, N.J., 1975.
- Zadeh, L.A. Fuzzy sets. *Information and Control*, 8 (1965), pp. 338-353.
- Zadeh, L.A. Fuzzy algorithms. *Information and control*, 12 (1968), pp. 94-102.

- Zadeh, L.A. Biological applications of the theory of fuzzy sets and systems. En L.D. Proctor (ed.): *Biocybernetics of the central nervous system*. Little, Brown and Co., Boston, Mass., 1969, pp. 199-212.
- Zadeh, L.A. Outline of a new approach to the analysis of complex systems and decision process. *IEEE Trans. Systems, Man, and Cyb.*, SMC-3 (1973), pp. 28-44
- Zadeh, L.A. Fuzzy logic and its applications to approximate reasoning. *Proc. IFIP Congress Information Processing 74*, North- Holland, Amsterdam, 1974, pp. 591-594.
- Zadeh, L.A. Fuzzy sets as a basis for a theory of possibility. *Fuzzy sets and systems*, 1 (1978), pp. 3-28.
- Zadeh, L.A. A theory of approximate reasoning. *Machine Intelligence*, 9. John Wiley, New York, 1979, pp. 149-194.
- Zadeh, L.A. The calculus of fuzzy if-then rules. *AI Expert*, 7, 3 (1992), pp. 22-27.
- Ziarko, W. (ed.) *Rough sets and knowledge discovery*. Springer- Verlag, Berlin, 1994.

Índice alfabético

Símbolos

0+, lógica 233

A

Abducción 192

Adquisición

de conceptos 388,390

del conocimiento 223,385

Alcance de un cuantificador 154

Alfabeto 33,53,152,559

Algebra de Boole 67

Algoritmo 399,411

de encadenamiento hacia adelante 228,232

de encadenamiento hacia atrás 228,233

genético 386,393

definición 411

presentación 399

propiedades 417

y máquina de Turing 403

y programas 401

Alto nivel, lenguaje de 614,621

Ambigüedad 579,638

Análisis

léxico 662,666,672

semántico 662,665

AND 95

Aplicabilidad, problema de 515

Aprendizaje 377,384,385

autómata con 377

definición 377,385

empírico 390

en máquinas 384

presentación 384

Arbol

definición 577

sintáctico 577

Asignación 160

Autómata 255,261

borroso 379

celular 382

con aprendizaje 377

de estructura variable 376

de pila 593

equivalente 278

equivalente de red de Petri 365

estocástico 373

finito 262

finito no determinista 596

limitado linealmente 592

minimización 297

monoide de 291

red de 383

Axioma 52,56,155

Axiomática, semántica 650

Axiomático, sistema 51

B

Backus, notación de 629
Bajo nivel, lenguaje de 614
Base de conocimientos 223
Base de datos 224
Base de datos deductiva 685
Biestable 312
BJ, diagramas 427
BNF 629
Borrosa
 función de transición 379
 lógica 196,206
 regla de inferencia 212
 relación 201,203
Borroso
 autómata 379
 subconjunto 198
Búsqueda
 exhaustiva 83
 heurística 221

C

Cadena 33,53
Cadena vacía 33,566,605
Cálculo 51
Canónica, forma 125
Cascada, metodología en 449
Caso de sustitución 177
Castigo, probabilidades de 378
Categoría sintáctica 662
Celular, autómata 382
Certidumbre, factor de 237
Ciclomático, número 541
Circuito lógico
 combinacional 96
 secuencial 310
Cláusula 78,172
 de Horn 81,176,627
 semántica 649
Clases 454
 lógica de 189
Cognoscitivo, enfoque 385,388
Compilador 615,661
 de compiladores 686
 generación de código 665
 optimización 665
Complejidad computacional 404,519
 caso peor 532
 cota inferior 532
 cota superior 532

definición 519
del software 405,537
espacial 523
exponencial 528
medidas 526
polinómica 528
presentación 404
temporal 523
 y máquina de Turing 520
Completitud 52,64,76,535
Comportamiento
 de entrada-estados 288,295
 de entrada-salida 277,295
Computabilidad 404
Computable, problema 403
Concatenación 34
Conceptualización 60,158
Conductista, enfoque 385
Conectiva 40,53,66
Conexionista, sistema 389
Conjunto
 borroso 198
 regular 340
Conocimiento
 adquisición de 223,385
 base de 223
 ingeniería del 223,224,384
Consistencia 52,64,76
Contador binario 293
Contradicción 64
Convivencial, programa 453
CORBA (Common Object Request Broker
 Architecture) 466
Cuantificador
 alcance 154
 existencial 144
 universal 144

D

D (biestable) 313
De Morgan, leyes de 59,68
Declarativa, programación 625
Declarativo, lenguaje 614,625
Deductiva, base de datos 685
Deductivo, razonamiento 30,73,167
Demostración 57,157
Denotacional, semántica 647
Derivación, relación de 560
Derivada de expresión regular 346
Desarrollo, entorno de 629
Descubrimiento 391

Diagrama
 BJ 427
 de Moore 263
 de flujo 424
 sintáctico 640
 Diseño descendente 433
 Disparo de red de Petri 362
 Dominio
 semántico 647
 sintáctico 647

E

Ejecución de red de Petri 363
 Ejemplar 177,248
 Encadenamiento
 hacia adelante 228,232
 hacia atrás 228,233
 Encapsulamiento 457
 Enlace dinámico 457
 Ensamblador, lenguaje 614,619,652
 Entorno 629,644
 Entrada-estados, comportamiento de 288,295
 Entrada-salida, comportamiento de 277,295
 Entrenamiento 385,387
 Equirrespuesta, relación 290,337
 Equivalencia 64
 de autómatas 278
 de gramáticas 561
 de sentencias 55,64,167
 Esquema de refuerzo 377
 Estado 257 261
 de red de Petri 362
 Estocástico
 autómata 373
 reconocedor 392
 Evaluación 61,161
 binaria 41,62,105
 Exclusión mutua 367
 Existencial, cuantificador 144
 Experto, sistema 222
 Explorador 662
 Expresión regular 341

F

Factor de certidumbre 237
 Finito, reconocedor 334
 Física del software 538
 Forma canónica 125
 Forma clausulada 78,172
 Formación de conceptos 391
 Formal, lógica 24

Fórmula atómica 143, 153
 Fórmula molecular 154
 Función 152,159,623
 computable 508
 de conmutación 102
 de salida 262
 de transición 262
 de transición borrosa 379
 de transición de red de Petri 364
 recursiva 508
 Funcional, programación 626

G

Genético, algoritmo 386,393
 Generación de código 665
 Gödel, números de 510
 Gramática
 ambigua 581
 de lenguaje de programación 631
 definición 559
 equivalencia 561
 generativa 568
 lenguaje generado por 561

H

Herencia 456
 Herramienta 224,628,644,685
 Heurística, búsqueda 221
 Heurístico 221,243

I

Identidad, lógica con 190
 Imperativa, programación 625
 Implicación
 estricta 191
 lógica 72,167
 material 43
 Imprecisión 236
 Incertidumbre 236
 Incompatibilidad, principio de 197
 Indecidible, problema 515
 Inferencia
 abductiva 192
 bayesiana 236
 borrosa 212
 imprecisa 196
 plausible 235
 regla de 73
 Inferencial
 proceso 76

- sistema 76
- Inferencias, motor de 223
- Ingeniería del conocimiento 223,224,384
- Instrucción
 - de máquina 615
 - de retorno 617
 - de salto 617
 - de salto a subprograma 617
- Intérprete 611
- Inteligencia artificial 31,219,384,569
- Interpretación 41,60,159
 - de red de Petri 366
 - pretendida 160

J

- JK (biestable) 313

K

- Karnaugh, tabla de 110

L

- Léxico, análisis 662,666,672
- Lectores y redactores 367
- Lenguaje 34,551,686
 - ambiguo 581
 - de alto nivel 614,621
 - de bajo nivel 614
 - de máquina 611,615
 - de programación 614
 - declarativo 614,625
 - ensamblador 614,619,652
 - fuelle 612
 - generado por una gramática 561
 - jerarquía de 565
 - macroensamblador 621
 - objeto 612
 - OOP 465
 - regular 595
 - tipo 0 y máquina de Turing 515
 - transportable 614
 - procesador de 612,652
 - tipos de 564
 - y autómatas 589
- Lex 686
- Ley 52,57
- Leyes de de Morgan 59,68
- Libre de contexto, gramática 565
- Línea de retardo 311
- Lingüística, aproximación 213
- Literal 53

- Lógica 29
 - 0+ 233
 - borrosa 196,206
 - con identidad 190
 - de clases 189
 - de predicados 141
 - de proposiciones 39
 - de relaciones 189
 - formal 29
 - implicación 72,167
 - modal 191
 - multivalorada 193
 - no monótona 192
 - probabilitaria 195
 - programación 627
 - puerta 94
 - temporal 193
 - trivalorada 194
- Lógico, circuito
 - combinacional 96
 - secuencial 310

M

- Máquina
 - aprendizaje en 384
 - de Mealy 264
 - de Moore 264
 - de programa almacenado 617
 - virtual 402,612
- Máquina de Turing 403,470
 - cálculo 476
 - composición 482
 - definición 470
 - descripción instantánea 474
 - diagrama de estados 477
 - diseño 486
 - esquema funcional 477
 - presentación 403
 - simulación por ordenador 491
 - simulador 500
 - universal 495
 - y algoritmos 403
 - y complejidad 520
 - y función computable 508
 - y lenguaje tipo 0 515
- M.A.P.S. (Methodology for Algorithmic Problem Solving) 499
- Macroensamblador, lenguaje 621
- Marcado de red de Petri 362
- Marco 248
- Mealy, máquina de 264

- Mensajes 455
- Metalenguaje 35,341,552
- Metodologías
 - de análisis orientado a objetos 461
 - de diseño orientado a objetos 461
 - en cascada 449
- Métodos 456
- Minería de datos 391
- Minimización de autómatas 297
- Modal, lógica 191
- Modelo 165
- Modus ponens 58,158
- Modus tollens 58,158
- Monádico, predicado 142
- Monoide de un autómata 291
- Moore
 - diagrama de 263
 - máquina de 264
- Motor de inferencias 223
- Multivalorada, lógica 193

N

- NAND 66,130,133
- Neurona 386
- Neuronal, red 386
- No monótona, lógica 192
- NOR 66,130,133
- NOT 95
- Notación de Backus 629
- NP, problemas 534
- Número ciclomático 541
- Números de Gödel 510

O

- Objetivos, procesador de 683
- Objetos
 - análisis orientado a 458
 - diseño orientado a 458
 - propiedades de 456
 - tecnología de 466
- OOP, lenguaje 465
- Operacional, semántica 645
- Optimización, fase de 665
- OR 95
- OR exclusivo 130
- Organigrama 426

P

- P-NP, problemas 534
- Paridad, detector de 267,291,322
- Perceptron 386

- Petri, red de 360
- Pila 593
- Poliádico, predicado 143
- Polimorfismo 457
- Pragmática 53,553
- Predicado 142
 - monádico 142
 - poliádico 143
- Predicados, lógica de 141
- Premio Turing 403
- Probabilístico, autómata 373
- Probabilidades
 - de castigo 378
 - de transición 374
- Probabilitaria, lógica 195
- Problema
 - computable 403
 - de aplicabilidad 515
 - de la deducción 419
 - indecidable 515
 - NP 534
 - P-NP 534
 - tamaño del 533
- Procedimiento 623
- Procesador
 - de lenguaje 612,652
 - de objetivos 683
 - de reglas 683
- Proceso inferencial 76
- Producción 559
 - regla de 224
 - sistema de 225
- Productores y consumidores 370
- Programa
 - análisis 448
 - codificación 448
 - convivencial 453
 - correcto 451
 - definición 424
 - diseño 448
 - estructurado 428, 432
 - limpio 427
 - robusto 453
- Programación
 - a gran escala 537
 - con Pascal y C 437
 - estructurada 423
 - orientada a objetos 453
 - entorno de 629,644
 - lenguaje de 614
 - tipos de 625
- Proposicional, variable 39,53

Proposiciones, lógica de 39
Prueba de Turing 220
Puerta lógica 94

R

Razonamiento 29,39,73
 deductivo 30,73
 válido 30,73,167
Reconocedor
 de Turing 590
 estocástico 392
 finito 334
 finito, análisis 344
 finito, síntesis 346
Rekursividad 576
Recurso abstracto
 definición 432
 presentación 402
Red
 de autómatas 383
 de Petri 360
 de Petri binaria 366
 de Petri, interpretación 366
 neuronal 386
 semántica 249
Refuerzo, esquema de 377
Refutación 85,180
Regla
 de escritura 559
 de inferencia 73
 de inferencia borrosa 212
 de producción 225
 de resolución 178
 procesador de 683
Regular
 conjunto 340
 expresión 341
 gramática 565
Relación 142,159
 borrosa 201,203
 de congruencia derecha 338
 de derivación 560
 equirrespuesta 290,337
 entre objetos 456
Relaciones, lógica de 189
Resolución
 impulsada por hechos 231
 regla de 178
Restrictiva, gramática 564
Retorno, instrucción de 617
Retropropagación 386

Reutilización 458
Robusto, programa 453

S

Salida, función de 262
Salto, instrucción de 617
Satisfacción 63,161
Secuencia de formación 54,153
Semáforo 367
Semántica 35,60,158,162,553,643
 cláusula 649
 red 249
 tipos de 645
Semántico
 análisis 662,665
 dominio 647
Sensible al contexto, gramática 564
Sentencia 54,154,621
 abierta 143, 155
 cerrada 143, 155
 equivalente 55,64,167
 válida 146,164
Seudoinstrucción 620
Sincronización 367
Sintáctica, categoría 662
Sintáctico, dominio 647
Sintagma 553
Sintaxis 35,53,152,553,637
Sistema
 axiomático 51
 conexionista 389
 de producción 225
 experto 222
 inferencial 76
 PM 56,155
Software
 complejidad del 405,537
 física del 538
SR (biestable) 312
Subprograma 617
Sumador binario 267,320
Sustitución 56,156,176
 caso de 177

T

Tabla
 de Karnaugh 110
 de transiciones 262
Tautología 64,164
Tecnología de objetos 466
Temporal, lógica 193

Teorema 52,57,157
Término 153
Tesis 52,57
Tipo de datos 624
Tipos de lenguaje 564
Traductor 612
Transición
 borrosa 379
 función de 262
 probabilidades de 374
 tabla de 262
Tripla O-A-V 234
Trivalorada, lógica 194
Turing (ver «máquina de Turing»)
 premio 403
 prueba de 220
 reconocedor de 590

U

Unificación 178
Universal, cuantificador 144
Universo del discurso 158

V

Válido, razonamiento 30,73,167
Variable
 booleana 104
 libre 154
Verdad, valores de 60
Virtual, máquina 402,612

Y

Yacc 686